

Analysis-Based Verification: A Programmer-Oriented Approach to the Assurance of Mechanical Program Properties

T. J. Halloran

May 27, 2010

CMU-ISR-10-112

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:

William L. Scherlis (advisor)

James D. Herbsleb

Mary Shaw

Joshua J. Bloch, Google, Inc.

Copyright © 2010 T. J. Halloran

Report Documentation Page		Form Approved OMB No. 0704-0188
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.		
1. REPORT DATE 27 MAY 2010	2. REPORT TYPE	3. DATES COVERED 00-00-2010 to 00-00-2010
4. TITLE AND SUBTITLE Analysis-Based Verification: A Programmer-Oriented Approach to the Assurance of Mechanical Program Properties		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER	
	5e. TASK NUMBER	
	5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University, Institute for Software Research, School of Computer Science, Pittsburgh, PA, 15213		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

There is a constant and insidious loss of design intent throughout the software lifecycle. Developers make design decisions but fail to record these decisions or their rationale. As a consequence, quality and maintainability of software suffer, since additional effort must be expended to recover and verify lost design intent prior to implementing even minor changes in the code. This is particularly challenging for concurrent code. Our vision is to capture and verify critical design intent through the use of fragmentary specifications supported by targeted verification tools that can function alongside debugging and testing tools in the practitioner's toolkit for software quality and maintainability. This thesis advances the idea of focused analysis-based verification as a scalable and adoptable approach to the verification of mechanical program properties. The main contribution of the research is the development of the concept of sound combined analyses, through which results of diverse low-level program analyses can be combined in a sound way to yield results of interest to software developers. The contribution includes the underlying logic of combined analysis, the design of the user experience and tool engineering approach, and field validation on diverse commercial and open source code bases. Building on prior work in semantic program analysis, this approach enables sound tool-supported verification of nontrivial narrowly-focused mechanical properties about programs with respect to explicit models of design intent. These models are typically expressed as code annotations, and can be used even when adopted late in the software lifecycle for real-world systems. In addition to providing a sound approach to combining fragmentary analysis results, the logic can support abductive inference of additional fragments of design intent. The proposed fragments that are deemed valid by the software developer can then be verified for consistency with code using an automated tool. The soundness of the logic for combined analysis is proved using an intuitionistic natural deduction calculus and other techniques. We validate our approach through the 9 field trials of a prototype tool that verifies properties related to multithreading and race conditions on a diverse sample of commercial, open source, and government code. In the majority of the field trials, this validation process included direct use of the prototype tool by disinterested professional developers and demonstrated that the tool performs useful verification and bug finding on full-scale production code.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT
unclassified

b. ABSTRACT
unclassified

c. THIS PAGE
unclassified

17. LIMITATION OF ABSTRACT

**Same as
Report (SAR)**

18. NUMBER OF PAGES

246

19a. NAME OF RESPONSIBLE PERSON

Keywords: analysis-based verification, annotation, assurance, annotation, formal methods, Java, program analysis, promises, quality assurance, quality assurance tools, sound composable analyses, specification, software engineering, software engineering tools, verification

Abstract

There is a constant and insidious loss of design intent throughout the software lifecycle. Developers make design decisions but fail to record these decisions or their rationale. As a consequence, quality and maintainability of software suffer, since additional effort must be expended to recover—and verify—lost design intent prior to implementing even minor changes in the code. This is particularly challenging for concurrent code. Our vision is to capture and verify critical design intent through the use of fragmentary specifications supported by targeted verification tools that can function alongside debugging and testing tools in the practitioner’s toolkit for software quality and maintainability.

This thesis advances the idea of focused analysis-based verification as a scalable and adoptable approach to the verification of mechanical program properties. The main contribution of the research is the development of the concept of sound combined analyses, through which results of diverse low-level program analyses can be combined in a sound way to yield results of interest to software developers. The contribution includes the underlying logic of combined analysis, the design of the user experience and tool engineering approach, and field validation on diverse commercial and open source code bases. Building on prior work in semantic program analysis, this approach enables sound tool-supported verification of non-trivial narrowly-focused mechanical properties about programs with respect to explicit models of design intent. These models are typically expressed as code annotations, and can be used even when adopted late in the software lifecycle for real-world systems.

In addition to providing a sound approach to combining fragmentary analysis results, the logic can support abductive inference of additional fragments of design intent. The proposed fragments that are deemed valid by the software developer can then be verified for consistency with code using an automated tool. The soundness of the logic for combined analysis is proved using an intuitionistic natural deduction calculus and other techniques. We validate our approach through the 9 field trials of a prototype tool that verifies properties related to multithreading and race conditions on a diverse sample of commercial, open source, and government code. In the majority of the field trials, this validation process included direct use of the prototype tool by disinterested professional developers and demonstrated that the tool performs useful verification and bug finding on full-scale production code.

Acknowledgments

One of my academic ancestors, Robert W. Floyd, used to say that he was planning to get a Ph.D. using the “green stamp method,” namely by saving envelopes addressed to him as ‘Dr. Floyd’. After collecting 500 such letters, he mused, a university somewhere in Arizona would probably grant him a degree. As a *ghost*, SCS’s colorful term for an *All But Dissertation* Candidate *In Absentia*, for six years I often wondered if the university somewhere in Arizona had updated this degree-granting method to include email as well as postal mail. I am fortunate indeed that my musings turned out to be unnecessary.

To Bill Scherlis, adviser and friend, for his support and guidance. Words just can’t convey my deep personal thanks to him and his family. I’ll take this opportunity to remind him that adviser is a *tenured* position.

To my world-class committee, Jim, Josh, and Mary, for their ongoing guidance and encouragement. Jim Herbsleb has been a constant source of encouragement for all aspects of this research—he is truly, as Burns put it, “*the gentleman an’ scholar*.” Josh Bloch helped to keep my focus on the realities of practice and provided timely and helpful feedback. I’ve reviewed more pages of Josh’s excellent books than are in this dissertation—unless any page with a Greek letter is counted as $[\pi]$ pages—in which case Josh is (way) ahead. Mary Shaw, first through her series of software engineering courses and later through her support of this research, has been instrumental in my development as a researcher and also as an educator.

To the Fluid research group, John Boyland, Edwin Chan, Aaron Greenhouse, Elissa Newman, and Dean Sutherland for their valuable feedback and support. In particular, Aaron Greenhouse, Edwin Chan, and Dean Sutherland were my “partners in crime” through prototype after prototype and field trial after field trial. Finally, last but not least, John Boyland demanded precision and cried “foul” to many a muddled idea. John reviewed the technical material in Chapter 2 several times and was unfailingly generous with his time and knowledge.

To Jonathan Aldrich, Nathan Boy, Konstantin (Cos) Boudnik, Rick Buskens, Dan Craigen, Dan Dvorak, George Fairbanks, Nick Frollini, David Garlan, Brian Goetz, Robert Graham, Greg Hartman, Connie Herold, Peter Homeier, Patrick Lardieri, Doug Lea, Larry Maccherone, Brad Martin, Todd McDonald, Don McGillen, Kenny Meyer, Mark Saaltink, David Wagner, all the organizations that participated in our field trials, and the CMU and AFIT software engineering groups for their support and valuable feedback.

To my wife Ellen, son Connor, and daughter Maggie (who was born two days before my first CMU exam—she’s 8-years old now!) for their constant support and encouragement—often of the *get finished* variety. I did.

Thank you!

Contents

1	Introduction	1
1.1	Vision	1
1.2	What is a mechanical program property?	2
1.3	The problem	2
1.3.1	Scalability and adoptability of verification and analysis tools	2
1.3.2	False positives and false negatives	4
1.4	Our approach: Analysis-based verification	7
1.4.1	Supporting verification	15
1.4.2	Supporting model expression	22
1.4.3	Supporting contingencies	29
1.5	Issues of adoption	33
1.6	Claims and contributions	36
1.6.1	The vision of the Fluid project	36
1.6.2	This thesis	37
1.6.3	Validation	39
1.7	Field trials in a nutshell	40
1.8	Case studies of constructing new aggregate analyses in a nutshell	43
1.9	Related work	43
1.10	Outline	45
2	Foundations	47
2.1	Introduction	47
2.1.1	A motivating example	48
2.1.2	Interpreting verification results	52
2.1.3	A running example	53
2.2	Promise logic	55
2.2.1	Syntax	55

2.2.2	Promise symbols	57
2.2.3	Proof theory	57
2.2.4	Models	58
2.3	Analysis results	59
2.3.1	Requirements for program analyses	62
2.4	Promise matching	63
2.5	Proof calculus for promise verification	64
2.5.1	Verification condition generation	67
2.5.2	Proof rules	68
2.5.3	The goal	69
2.5.4	Including verification conditions	70
2.5.5	Linking to promise logic	70
2.5.6	Composing results	71
2.5.7	Handling recursion	73
2.5.8	Handling “mixed” prerequisite assertions	77
2.6	Semantics	78
2.7	Soundness	81
2.8	Not a Hoare logic	86
2.9	Conclusion	87
3	Realization	89
3.1	Introduction	89
3.2	Programmer–tool interaction	91
3.2.1	Tool suggestions to get started	91
3.2.2	Adding annotations	92
3.2.3	Tool output	92
3.3	Tool architecture	97
3.3.1	Java code representation	97
3.3.2	Promise “scrubbing”	97
3.3.3	Analysis of the “forest”	99
3.3.4	Promise matching	100
3.3.5	Promise verification	100
3.3.6	Querying the tool results	100
3.4	The drop-sea proof management system	101
3.4.1	Representing analysis results	101

3.4.2	Representing recursion	102
3.4.3	Representing unmatched proposed promises	104
3.4.4	Representing contingencies	106
3.4.5	Computing verification results	106
3.4.6	Truth maintenance	117
3.5	Scoped promises	118
3.5.1	Avoiding repetitive annotation	118
3.5.2	Team modeling with assumptions	123
3.5.3	Defining a payload	125
3.5.4	Defining a target	125
3.5.5	Programmer vouches	126
3.6	Trusted promises	130
3.7	Related work	130
3.8	Conclusion	131
4	Field trials	133
4.1	Introduction	133
4.1.1	Organizations visited and code examined	133
4.1.2	Summative evaluation	136
4.1.3	Formative evaluation	137
4.2	Methods	138
4.2.1	Participants	138
4.2.2	Agenda	140
4.2.3	Facilities	142
4.2.4	Preparation	143
4.2.5	Gathering data	143
4.3	Analysis	143
4.4	Discussion	154
4.5	Threats to validity	161
4.6	Conclusion	162
5	Case studies: Constructing aggregate analyses	163
5.1	Introduction	163
5.2	Case study: Thread coloring	163
5.3	Case study: Static layers	166
5.4	Conclusion	168

6	Specification and verification of static program structure	169
6.1	Introduction	169
6.2	Specification	170
6.2.1	A simple model	171
6.2.2	A layered model	173
6.2.3	Composing layered models	176
6.2.4	Checking that layered models are well-formed	178
6.2.5	Restricting allowed references	180
6.3	Verification	181
6.3.1	Checking for well-formed models	181
6.3.2	Checking model–code consistency	183
6.4	Not a module system	186
6.5	Conclusion	186
7	Validation	187
7.1	Sound combined analyses	187
7.2	Cost-effectiveness analysis	190
7.2.1	Cost	190
7.2.2	Benefit	190
7.2.3	Alternative approaches	191
8	Conclusion	193
8.1	Summary of contributions	193
8.2	Looking forward	195
8.2.1	Accommodating negative analysis results	195
8.2.2	Supporting query-based modeling	195
8.2.3	Hybrid analysis-based assurance tools	196
A	JSure modeling guide	197
A.1	Lock policy	198
A.1.1	A straightforward model	198
A.1.2	Extending the model: Caller locking	199
A.1.3	Extending the model: Aggregating arrays and other objects	200
A.1.4	Constructors	202
A.1.5	Thread-safe objects	203
A.1.6	Regions and locks	204

A.1.7	Intrinsic or <code>java.util.concurrent</code> locks	206
A.1.8	Declaring multiple locks	208
A.1.9	Returning locks	208
A.1.10	Policy locks	209
A.2	Method effects	211
A.2.1	Effects and constructors	212
A.3	Unshared fields	214
A.3.1	Borrowed references	214
A.3.2	Supporting borrowed with method effects	214
A.4	Thread effects	216
A.5	Scoped promises	217
A.6	Programmer vouches	219
B	Extended Backus-Naur form	221
C	Glossary	223

Introduction

“Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it.” — C. A. R. Hoare [61]

1.1 Vision

Formal program specification and verification has yet to find widespread adoption in mainstream practice. Despite a lack of practical verification tools, programmers have nonetheless felt the need to express and verify additional specification information regarding their intent for the programs they develop through models, documentation, and other specifications. Often these extra-linguistic models and specifications are of sufficient utility and value that they influence the design of later generations of programming languages to provide more expressiveness with respect to both higher level abstractions and design intent. One historical example is the introduction of explicit strong typing into programming languages, such as Ada 95, Java, and C#, replacing informal notations such as Microsoft’s “Hungarian notation” [100].

Our vision is to accelerate progress towards the use of specification and verification tools alongside debugging and testing tools in the practitioner’s toolkit for software quality. We place particular focus on narrowly-targeted “extra-functional” or “mechanical” requirements (defined below) related to quality attributes. We do this because of the greater possibility, within these constraints, of achieving scalable and adoptable verification capabilities. The overall vision encompasses sound static, heuristic static, and dynamic analysis of the consistency of code with programmer-expressed models of design intent, done at scale and in a way that could eventually result in tools usable by working software developers. The models should be expressible in a variety of ways, for example, annotations in source code and stand-off annotations, such as for library code. The vision also includes aggregating the results of multiple targeted low-level analyses in order to obtain higher-level conclusions more directly usable by software developers. For example, analysis of correct locking in concurrent Java programs relies on a combination of effects analysis, alias analysis, and several special purpose analyses, each of which may be supported by particular annotations.

1.2 What is a mechanical program property?

What do we mean by narrowly-targeted “extra-functional” or “mechanical” requirements related to quality attributes? We use these terms to direct the focus of specification away from what functionality the program implements toward the “mechanism” by which the program achieves that functionality.

Instead of asking the programmer to explicitly express a full functional specification, we ask the programmer to record specific attribute-focused design intent related to how the program does its work. These properties can often be expressed using succinct specification fragments, sometimes just one line of additional annotation text associated with the program. These properties may be essential to correct or safe execution, but difficult to manually check. We refer to these difficult to check, extra-functional properties as *mechanical program properties*. These properties can be considered part of a broader set of overall program invariant properties.

1.3 The problem

1.3.1 Scalability and adoptability of verification and analysis tools

There is a constant and insidious loss of *design intent* throughout the software lifecycle. Developers make design decisions but fail to record these decisions or their rationale. As a consequence, maintainability and quality of software suffer, since additional effort must be expended to recover lost design intent prior to implementing even minor changes in the code. Traditional formal specification and program verification techniques, such as PVS [32, 90, 91], Larch [57, 107], VDM [16, 17], and Z [94], provide a partial solution focusing generally on functional properties. These tools tend to require an up-front commitment by developers to produce a formal specification and maintain that specification throughout system evolution and maintenance. The up-front nature of this commitment and the cost and difficulty of delivering on it limits the practicability of these techniques even when introduced at the beginning of the software lifecycle. A commitment to any one of these techniques for a large code-base can be expensive, in terms of up-front cost and schedule, but it provides the strategic benefit of increased code quality. In practice, a manager of a project with an uncertain outcome/business value may be unwilling to pay these immediate costs for a long-term quality improvement that is difficult to measure. The reality is that *large* software systems, with *little* documentation, could benefit from formal specification and program verification techniques, but these systems are in the worst possible position to adopt the techniques.

Throughout the lifetime of a software system people join and leave its development and maintenance team. As time passes informal design information, often due to neglect, becomes out of date and inconsistent—leaving *source code* as the only authoritative system artifact. Maintainability suffers because code does not reveal all the design intent behind it. Maintainability and new development, both are further complicated by the increasing use of large and complex libraries and frameworks—each with its own set of usage constraints not enforced through checks made by compilers and loaders and each evolving independently of the systems that depend upon them. Quality suffers when design intent is tacit or informally expressed, since programmers may give incorrect interpretations when intent is expressed informally, or they may make wrong guesses when intent is tacit.

There are, in fact, good reasons why we are experiencing this problem. It is difficult in our software languages, models, and tools to focus in a rigorous way on specific aspects of design intent relating to particular quality attributes. An early and specialized example of focused automated analysis is provided by the SLAM tool at Microsoft, which targets consistency of device driver code with protocol requirements associated with the Windows device driver API. This is a very narrow “focus” of intent, but it turns out to be particularly significant in reducing the frequency of blue screen failures of the Windows OS [8]. This tool uses model checking in a specialized way.

An advantage of this overall approach of focusing on narrow aspects of design intent is that the associated verification task can often be more scalable, more computationally efficient, and more parsimonious in its requirements of explicit specification or design intent. Additionally, these targeted analyses can often be usefully composed to deliver more aggregated results, such as in the example of locking analysis alluded to above. Indeed, there is a large variety of particular quality attributes that could potentially be addressed in this focused way. Mitre’s Common Weakness Enumeration (CWE) extensive taxonomy is an example of an inventory of such quality attributes¹.

This is part of the motivation for our vision, which is to build on the idea of targeted analysis, as evidenced in early specialized tools, to create a general framework that addresses issues of scale, adoptability, composability of software components, and aggregation of multiple constituent analyses. In our approach, we express and capture the focused intent formally and use analysis tooling and our novel proof management capability to assure that our implementations are faithful to that intent. One of the key challenges related to adoptability is the means by which developers can keep this documentation of intent consistent with the as-built reality of a system as both evolve.

What criteria must be considered to facilitate the acceptance of verification tools into software engineering practice? This is not an easy question to answer definitively given the wide diversity of today’s software engineering practices; however, we identify the following to be key:

- **Scale and composition:** Tools should support “separate verification” in a manner similar to separate compilation. Composition is key to the ability to scale up, in terms of code size, to real-world software systems.
- **Process compatibility:** Tools should operate within familiar tools and processes used by working developers in practice. In particular, it is important to avoid mandating that an organization has to fundamentally change their tools or processes (or people) simply to use an approach.
- **Support programming teams:** Modern software is typically decomposed into many components and may depend upon many libraries. Each component or library being developed and maintained by its own team of programmers. Teams (or teams of teams) should have a principled approach to collaborate on models of properties.
- **Gentle-slope ROI:** Tools should allow small increments of specification to yield immediate results—avoiding an “all-at-once” specification approach. To offer a “gentle slope” (this term due to Michael Dertouzos) with regard to return on investment (ROI)—increments of effort in adopting an approach yield increments of impact [66]. The

¹<http://cwe.mitre.org/>

		Actuality	
		Fault	No fault
Tool says	Fault	<i>true positive</i>	<i>false positive</i>
	No fault	<i>false negative</i>	<i>true negative</i>

Figure 1.1: The two types of errors that can occur in static analysis tools. In the case of a *false positive* the tool reports a bug that the program doesn't contain. In the case of a *false negative* the code contains a bug that the tool doesn't report. A tool is considered *sound* if it produces no false negatives (for a given set of assumptions) [27].

benefit should be as immediate and tangible as possible and to the developer and team, rather than (or, in addition to) deferred and more diffused across the organization.

- **Late and early lifecycle adoption:** Tools should be feasibility adoptable at any point in the software engineering lifecycle. When adopted early in the lifecycle the up-front commitment must be kept low. The approach should also facilitate late-lifecycle adoption for real-world software systems.
- **Attribute focus:** Tools should focus on particular quality attributes, which we have referred to as mechanical, to keep specification requirements frugal and to allow the associated verification task to be more scalable in terms of computational efficiency and concurrent development effort by team members.
- **Familiar expression:** Models of properties should be expressed tersely and using terminology already familiar to programmers.

1.3.2 False positives and false negatives

Recent work in heuristics-based static analysis tools has been successful in uncovering a large number of defects in real-world code. Tools such as FindBugs [62], PMD², and MC [42, 28, 6]³ are finding widespread use in practice. These tools are able to scan large bodies of existing code quickly. They support adoption late in the software lifecycle because they require no up-front commitment. They scan a codebase and report to the tool user any instances of known “bug patterns” [4] or violations of best practice [18, 82] that the analysis can find. For example, the FindBugs tool discovered an infinite recursive loop within the `AnnotationTypeMismatchException` class of the Java standard library which was fixed in the next release⁴.

```
private final String foundType;

public String foundType() {
    return this.foundType();
}
```

²<http://pmd.sourceforge.net>

³MC is the research basis for the Coverity Static Analysis tool (<http://www.coverity.com/>).

⁴http://bugs.sun.com/view_bug.do?bug_id=6179014

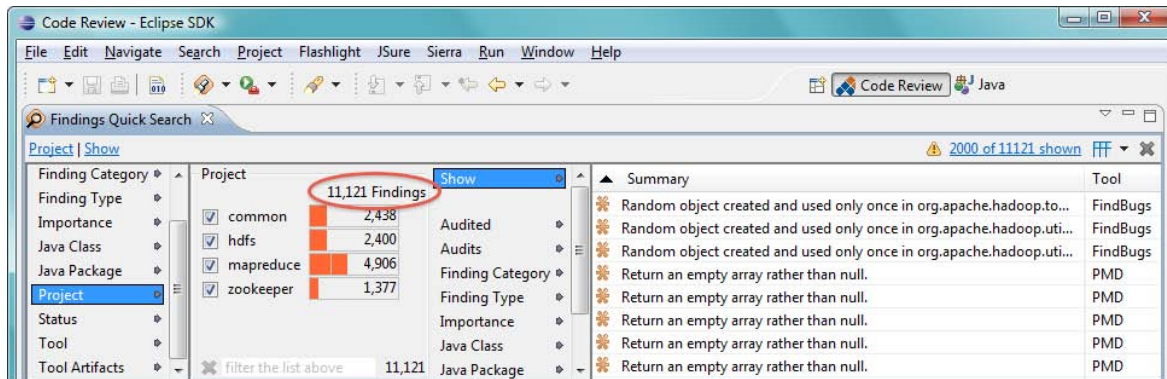


Figure 1.2: A scan of four Apache Hadoop subprojects (Hadoop Common, HDFS, MapReduce, and ZooKeeper) by FindBugs and PMD. Hadoop is a widely used “cloud” computing platform. The scan reported 11,121 findings (3,380 from FindBugs, 7,628 from PMD, and 113 from both tools). This overwhelming number of findings, a large percentage of which are *false positives* (the tool reports bugs that the program doesn’t contain), is difficult for tool users to address.

The problem of false positives

A limitation of heuristics-based static analysis tools is that they can produce a large number of findings, many of which are incorrect reports about bugs that the program doesn’t contain. This type of error, in the parlance of diagnostic tests, is referred to as a *false positive* as shown in Figure 1.1. For example, Figure 1.2 shows a scan of four Apache Hadoop subprojects, roughly 200 KSLOC of Java code, by FindBugs and PMD which produced 11,121 findings. The time commitment required to separate out false positive findings from findings that indicate a real defect in the code is significant. At this scale, just prioritizing findings and planning response actions is daunting (even if there were no false positives). Having to assign a programmer to perform this task each time the tool is run on the code would render the tool impractical for all but very small codebases.

Several techniques have been introduced to help mitigate the problem of false positives in heuristics-based static analysis tools. These include the matching of findings across scans of the code [101] so that once a finding is “marked” as a false positive by a programmer it will no longer be reported to the tool user, the use of statistical techniques to infer which findings are the most likely to indicate real bugs in the code [43], the categorization of tool findings for (web or graphical) user interfaces that allow filtering and querying by the tool user, and the annotation of *design intent* by the programmer into the code. Our interest is in the last.

Many of these tools provide an annotation to suppress a particular type of finding at a location in the code. The programmer is, in effect, avoiding a false positive report by telling the tool that this code is intended to be an exception to the heuristic rule. For example, the `@SuppressWarnings` annotation below suppresses a finding from the PMD tool about the method `DoSomething` violating the Java convention that method names begin with a lowercase letter.

```
@SuppressWarnings("PMD.MethodNamingConventions")
public void DoSomething() { ... }
```

While pragmatic, we view this approach as undesirable because it has the potential to litter

the code base with annotations of tool-specific semantics. Indeed, the semantics of a particular annotation may vary depending upon which version of an analysis tool is being used to scan the code. Further, if you also run FindBugs on your code you have to add another tool-specific annotation to suppress the same warning from that tool⁵.

```
@edu.umd.cs.findbugs.annotations.SuppressWarnings("NM_METHOD_NAMING_CONVENTION")
@SuppressWarnings("PMD.MethodNamingConventions")
public void DoSomething() { ... }
```

A better approach, implemented by FindBugs to help reduce false positive findings about null pointer defects [63], is illustrated below.

```
public @NonNull String convert(@NonNull Object o) { return o.toString(); }
```

The first `@NonNull` annotation indicates that this method is intended not to return a null value. The second indicates that callers are assumed to ensure that the reference they pass as the parameter `o` is non-null. Unlike the `@SuppressWarnings` annotation, the semantics of the `@NonNull` annotation are not (or at least have the potential to not be) tool- or version-specific and they precisely constrain the program’s implementation.

The problem of false negatives

FindBugs uses annotations with precise semantics, such as `@NonNull`, to help its ability to report potential code defects with a low level of false positives. However, today’s heuristics-based static analysis tools only give the programmer “bad news,” in the sense that the best result the tool can give is, “I didn’t find anything wrong.” This is because heuristic static analysis tools are typically *unsound*. This means, in the terminology defined in Figure 1.1, that the tools attempt to reduce false positives at the cost of letting some false negatives “slip by.” Prior to the introduction of annotations with precise semantics, this was a necessary compromise to make these tools practicable. This is analogous to the precision/recall trade-offs in information retrieval. The impact of too much emphasis on false positives (*i.e.*, on precision) is described humorously by Chess and McGraw in [27]:

“The static analysis crowd jokes that too high a percentage of false positives leads to 100 percent false negatives because that’s what you get when people stop using the tool.”

In fact, while the “holy grail” for many years has been to reduce false positives, it can be argued that there has been too little discussion regarding false negatives. One of the challenges of false negatives is understanding the “actual condition” as described in Figure 1.1—it can be difficult to ascertain what defects are actually not found in a meaningfully-sized code base. Generally speaking, false negatives can mislead an unwary tool user into a sense of security if they misinterpret the meaning of the results. When the tool outputs no results it is telling the user, as we noted above, “I didn’t find anything wrong.” *The programmer, however, wants*

⁵FindBugs is not able to use the `java.lang.SuppressWarnings` annotation because that annotation has only source retention (*i.e.*, annotations are discarded by the compiler) and FindBugs performs its analysis on Java bytecode (the output of the compiler). The `java.lang.SuppressWarnings` annotation was intended for use by the Java compiler. The source retention of the `java.lang.SuppressWarnings` annotation may be changed in a subsequent Java release to better support bytecode-based static analysis tools.

to know the answer to the more general question, “Is this design intent fully consistent with my code?” This is a question that can be answered only by verification or sound analysis.

1.4 Our approach: Sound combined analyses for analysis-based verification

Another way to state our vision regarding focused or narrowly-targeted specification and verification is that we can achieve practicable verification capability for a useful range of focused quality attributes. We demonstrate feasibility of this vision using an Eclipse-based prototype tool, which we call JSure, in the case of a number of (primarily) concurrency-related attributes, many of which are prior work. Figure 1.3 lists the focused quality attributes that can be verified by the JSure prototype analysis-based verification tool and the set of low-level constituent analyses used by the tool. Our key idea is that there is a collection of theory and engineering that makes the verification of highly-focused programmer expressed design intent (e.g., annotations with precise semantics) feasibly adoptable by practicing programmers.

We present the contributions of this thesis more rigorously in Section 1.6. In Section 1.6 we also clearly differentiate our work from the work of other members of the Fluid project at Carnegie Mellon University over the past decade. In this section, however, we briefly introduce our approach and then present a “tour” of its features. We describe BoundedFIFO as a non-trivial running example that we employ to showcase our work. We present how our approach supports verification of “promises” about code by examining the limitations of reporting sound analysis results in a manner similar to a compiler and sketch how we overcome these limitations. We use the term *promises*, as introduced by Chan, Boyland, and Scherlis in [26], to refer to annotations with precise semantics—highlighting that each one “promises” something about the behavior of the program. We continue by introducing two techniques, proposed promises and the scoped promise, @Promise, that assist the tool user with model expression. We end the section with a description of how our approach allows several unverified contingencies to exist in a chain of evidence about a promise.

Sound combined analyses

Sound combined analyses for analysis-based verification is a tool-supported approach to managing mechanical properties within a software system by providing composable model-based verification of these properties. Our approach creates verification results by combining fragmentary analysis results from multiple underlying analyses—linking together “chains” forged from small “links” of evidence reported by constituent analyses about a software system. A multitude of these chains of evidence, sometimes interconnecting and sometimes not, is formed for each program examined. Fundamentally, a “chain of evidence” is a proof of a theorem relating a programmer-expressed model of design intent with source code. The soundness of these proofs, and thus any verification results provided by our Java-oriented prototype tool, are contingent upon several assumptions:

- The Java compiler and virtual machine are faithful to the Java language semantics [52].
- The Java virtual machine ensures that consistency is maintained between the code that is verified and the code that is loaded at runtime. For Java, in particular, this means

Mechanical program property	Verifying analyses	Developer
Lock use (concurrency)	binding context effects upper bounds lock policy may equal must-hold lock must-release lock non-null reference uniqueness	Greenhouse [53]
Thread use (concurrency) also called “thread coloring”	binding context color constraint color constraint inference color constraint inheritance effects upper bounds may equal static call graph uniqueness	Sutherland [103]
Prohibiting new threads (concurrency)	thread effects	Halloran [55]
Modules (static program structure)	binding context effects upper bounds may equal module boundary effects module visibility referenced types uniqueness	Sutherland [103]
Layers (static program structure)	layers are ordered referenced types	Halloran (Ch. 6)

Figure 1.3: A list of the program properties supported by the JSure prototype tool and the member of the Fluid project research group who developed its specification language and verifying analyses. Descriptions of the verifying analyses are found in the references to prior work (or chapters of this document) provided in the table. The specification language for the listed mechanical program properties is summarized below in Figure 1.7 and Figure 1.8.

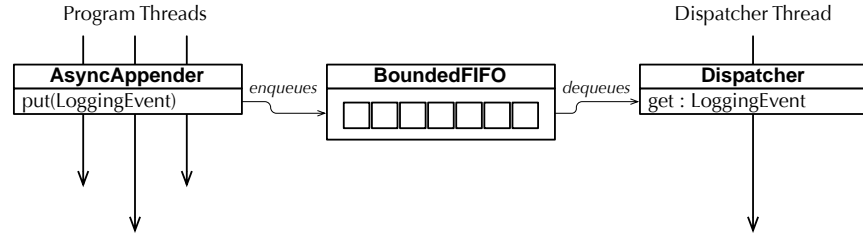


Figure 1.4: A sketch of the use of the `BoundedFIFO` class in a programmer performing logging using Apache Log4j. Multiple program threads share the buffer with a dispatcher thread.

that all classes loaded at runtime have been analyzed by the tool⁶.

- Any contingencies or gaps vouched for by the programmer, called “red dots” in the tool, do not, in fact, violate any models at runtime.

In general, models of programmer design intent (about the program properties listed in Figure 1.3) and their corresponding proofs of consistency are partial—this enables a programmer to move stepwise toward establishing some kind of verification result. When our approach fails to verify model–code consistency there can be several reasons: (1) the model is wrong, (2) the model is incomplete (perhaps more is needed to enable analysis to succeed), (3) the source code is wrong, (4) the inconsistency is intended, or (5) the analysis algorithms, due to limitations, have insufficient information to provide a result. The trade-off being made by a programmer using our approach is a loss of implementation flexibility for a gain in assurance.

Our approach to analysis-based verification is *not* meant to replace heuristic analysis, but rather to complement it—sound analyses are generally narrowly targeted and we suggest that they can be realized only for particular quality attributes, while heuristic analyses may cover a much broader range of quality attributes, most of which cannot yet be addressed through sound analysis techniques. Both approaches are intended to scale to large real-world code bases.

Example: BoundedFIFO

The `BoundedFIFO` class, taken from Apache Log4j⁷, was used by Greenhouse as an example of the specification and analysis techniques to verify lock use developed in [53]. This class buffers logging events shared between program threads and a dispatcher thread. Figure 1.4 illustrates this interaction. The use of `BoundedFIFO` helps to shield the program from logger latencies—the program thread enqueues a logging event into the buffer and returns, allowing the program to proceed with minimal interruption. The dispatcher thread removes events from the buffer and handles them (*e.g.*, writes them out to a disk file). The implementation

⁶Pragmatically, for many systems, all of a program’s classes can be checked by the tool or non-checked classes can simply be trusted. Such systems, by accepting the risk of a runtime error or failure, gain benefit from the tool without change to the runtime environment of the system, which may be considered a greater risk by the system’s programmers. Systems requiring a higher degree of assurance, such that they are unwilling to risk runtime violations, can develop a custom classloader to mandate that all classes loaded at runtime for a program are those that have been statically assured.

⁷<http://logging.apache.org/log4j/>

of `BoundedFIFO` uses locking to ensure thread safety. The annotations to the `BoundedFIFO` code, shown boxed in Figure 1.5, precisely document its locking policy for its intended use within the `Log4j` code. The semantics of the annotations, which we also refer to as *promises*, are explained in Appendix A, however, to make the presentation self-contained, we sketch their meaning below.

The `@RegionLock("FIFOLock is this protects Instance")` promise at line 1 specifies that reads and writes to any of the object’s state (the five fields declared in the class that are referred to as `Instance` in the promise syntax) must occur within a block synchronized (*i.e.*, holding the lock) on the object (*i.e.*, `this`). `FIFOLock` is an arbitrary name chosen by the programmer for this assertion.

The `@Unique` promise at line 4 asserts that the reference to the array held by the field `buf` is unique—no aliases to this array are allowed. The `@Aggregate` annotation includes the state of the array referenced by `buf` into `Instance`. Because `Instance` is protected by a lock, the array referenced by `buf` is as well. The `@Aggregate` annotation defines a portion of the `Instance` region for `BoundedFIFO`. It is either well-formed or not—but it does not constrain the program’s implementation and, therefore, no analysis results are reported about it. (Checking that annotations are well-formed is done via a process we call “scrubbing” which is described in Section 3.3.2.)

A `@RequiresLock("FIFOLock")` promise is annotated on each of the seven methods declared in `BoundedFIFO`. These promises specify that clients of a `BoundedFIFO` instance acquire the lock on that instance before invoking any methods on the object. For example, client code used to enqueue a logging event into a `BoundedFIFO` instance is shown in Figure 1.6.

The `@Unique("return")` promise at line 10 on the `BoundedFIFO` constructor asserts that the receiver under construction is not aliased during object construction, *i.e.*, that the reference to the object “returned” by this constructor is unique. In concurrent Java code, objects are typically confined to a single thread while they are constructed (*i.e.*, to the thread that invokes the `new` expression) then safely published to other threads in the program. If we can verify that the object is thread-confined during construction we can allow read and writes of its protected state during construction without holding the lock. The `@Unique("return")` promise verifies that the protected state of `BoundedFIFO` is thread-confined during construction because, in the Java programming language, threads can only communicate through fields and this assertion prohibits an alias to the object under construction from being written into a field (and, therefore, possibly making the object under construction visible to other threads).

We note that, by any measure, `BoundedFIFO`, is far from exemplary Java code. The field and method names are non-standard with respect to the conventions espoused by the standard Java collections library, and not even internally consistent. What they call `numElts` should be called `size`; what they call `size` and `maxSize` should be called `capacity`. Some accessor methods, *e.g.*, `getMaxSize()`, use the “get” convention, while others, *e.g.*, `length()`, don’t. The `put` method wastes its return value: if instead of `void`, it returned `true` if it succeeded the class would be much easier to use and less error prone.

A limitation of the annotations to `BoundedFIFO` shown in Figure 1.5 is that they imply that the class cannot be used without locking, which isn’t quite true. `BoundedFIFO` can safely be used by a single thread without locking. Our annotations reflect the specific use of this class within the `Log4j` code base. In this case, we deem this sufficient, because, for the reasons


```

1  @RegionLock("FIFOLock is this protects Instance")
2  public class BoundedFIFO {
3
4      @Unique
5      @Aggregate
6      LoggingEvent[] buf;
7
8      int numElts = 0, first = 0, next = 0, size;
9
10     @Unique("return")
11     public BoundedFIFO(int size) {
12         if (size < 1) throw new IllegalArgumentException();
13         this.size = size;
14         buf = new LoggingEvent[size];
15     }
16
17     @RequiresLock("FIFOLock")
18     public LoggingEvent get() {
19         if (numElts == 0) return null;
20         LoggingEvent r = buf[first];
21         if (++first == size) first = 0;
22         numElts--;
23         return r;
24     }
25
26     @RequiresLock("FIFOLock")
27     public void put(LoggingEvent o) {
28         if (numElts != size) {
29             buf[next] = o;
30             if (++next == size) next = 0;
31             numElts++;
32         }
33     }
34
35     @RequiresLock("FIFOLock")
36     public int getMaxSize() { return size; }
37
38     @RequiresLock("FIFOLock")
39     public int length() { return numElts; }
40
41     @RequiresLock("FIFOLock")
42     public boolean wasEmpty() { return numElts == 1; }
43
44     @RequiresLock("FIFOLock")
45     public boolean wasFull() { return numElts + 1 == size; }
46
47     @RequiresLock("FIFOLock")
48     public boolean isFull() { return numElts == size; }
49 }

```

Figure 1.5: Java code for the BoundedFIFO class from Apache Log4j after adding the locking and uniqueness annotations (shown in boxes) required to assure its locking policy.

```

public class AsyncAppender {

    private final BoundedFIFO fifo;

    AsyncAppender(BoundedFIFO value) {
        fifo = value;
    }

    void put(LoggingEvent e) {
        synchronized (fifo) {
            while (fifo.isFull()) {
                try {
                    fifo.wait();
                } catch (InterruptedException ignore) { }
            }
            fifo.put(e);
            if (fifo.isEmpty()) {
                fifo.notifyAll();
            }
        }
    }
    ...
}

```

Figure 1.6: Elided Java code for the `AsyncAppender` class used to enqueue a logging events from program threads into a `BoundedFIFO` instance that is shared with the dispatcher thread.

enumerated above, `BoundedFIFO` is a poor candidate for more widespread use.

`BoundedFIFO`, with all its problems, is typical of code we have sometimes encountered in the field (Chapter 4). If our approach is to be feasibly adoptable late in the software lifecycle it has to be able to deal with imperfect code like `BoundedFIFO`. Many times we observed that programmers were loath to change any aspect of (what they deemed to be) working code. In other cases, however, the use of the JSure prototype analysis-based verification tool on imperfect code exposed design problems within a class or, worse, an entire software system. This caused the focus of a field trial to move from running the tool to worried programmers sketching diagrams on a whiteboard.

The promises described above and the program analyses used to verify them are prior work. Analyses designed to support this composable annotation-based approach have been developed by members of the Fluid project at Carnegie Mellon University over the past decade. The lock policy promises, *e.g.*, `@RegionLock` and `@RequiresLock`, were developed by Greenhouse in support of lock analysis [53]. The region concept, *e.g.*, the `Instance` region and the `@Aggregate` promise, were developed by Greenhouse and Boyland [54]. The alias promise, `@Unique`, and associated uniqueness analysis were developed by Boyland [20, 21]. A list of the promises supported by the JSure prototype tool is given in Figure 1.7 and Figure 1.8. This list briefly describes the meaning of each annotation and indicates who developed it. Annotations that are contributions of this thesis are indicated in (blue) italics.

Annotation	Description (Developer)
<code>@Aggregate</code>	Declares that regions of the object referenced by this field are to be mapped into regions of the object that contains the field to which this annotation is applied. (Greenhouse, Boyland)
<code>@AggregateInRegion</code>	Declares that the <code>Instance</code> region of the object referenced by this field is to be mapped into a named region of the object that contains the field to which this annotation is applied. (Greenhouse, Boyland)
<i><code>@AllowsReferencesFrom</code></i>	Constrains the set of types that are allowed to reference the annotated program element. (Halloran)
<i><code>@Assume</code></i>	Declares a promise that is assumed about a portion of the system. (Halloran)
<i><code>@AssumeFinal</code></i>	Declares that the field or parameter to which this annotation is applied should be treated as if it is declared <code>final</code> , despite the fact that it is not. (Halloran)
<code>@Borrowed</code>	Declares that the parameter or receiver to which this annotation is applied does not receive any new aliases during execution of the method or constructor. (Boyland)
<code>@Color</code>	Constrains the annotated method to be callable only from a thread that satisfies a boolean expression over thread role names declared using <code>@ColorDeclare</code> . (Sutherland)
<code>@ColorDeclare</code>	Declares a named thread role, referred to as a <i>thread color</i> . (Sutherland)
<code>@Grant</code>	Declares that a particular thread role (<i>i.e.</i> , a named thread color) is granted to a thread. (Sutherland)
<code>@GuardedBy</code>	The field or method to which this annotation is applied can only be accessed when holding a particular lock, which may be a built-in (synchronization) lock, or may be an explicit <code>java.util.concurrent.Lock</code> . (Goetz, <i>et al.</i> [51])
<code>@Immutable</code>	The class to which this annotation is applied is immutable. (Goetz, <i>et al.</i> [51])
<code>@IncompatibleColors</code>	A global constraint on the program that at most one of the thread roles may be taken by a particular program thread at the same point in time. (Sutherland)
<i><code>@InLayer</code></i>	Declares that the annotated type is part of the named layers. (Halloran)
<code>@InRegion</code>	Declares that the field to which this annotation is applied is mapped into the named region. (Greenhouse, Boyland)
<i><code>@Layer</code></i>	Declares a named layer as well as the type set that types in the layer may refer to. (Halloran)
<code>@MaxColorCount</code>	Declares that a particular thread role may be associated to 0, 1, or n program threads. (Sutherland)
<i><code>@MayReferTo</code></i>	Constrains the set of types that the annotated type is allowed to reference. (Halloran)

Figure 1.7: (A through Ma) An alphabetical list of the annotations supported by the JSure prototype tool. Each annotation includes a brief description and the (parenthesized) name of its developer(s). Italicized annotations (also in blue) are contributions of this thesis. The majority of these annotations were developed by members of the Fluid project (with references to this prior work listed in Figure 1.3), however, the tool also supports the Goetz, *et al.* annotations from the book *Java Concurrency in Practice* [51]. The annotations are further described, with examples of their use, in Appendix A.

Annotation	Description (Developer)
<code>@Module</code>	Declares that the compilation unit to which this annotation is applied is part of the named module. (Sutherland)
<code>@NotThreadSafe</code>	The class to which this annotation is applied is not thread-safe. (Goetz, <i>et al.</i> [51])
<code>@NoVis</code>	Declares that the annotated type, field, method, or constructor is not of the exported interface of the module it is contained within. Overrides any default visibility, for example, it could be used on a method to override a <code>@Vis</code> annotation on a type. (Sutherland)
<code>@PolicyLock</code>	Declares a new policy lock for the class to which this annotation is applied. (Greenhouse)
<i><code>@Promise</code></i>	Declares a promise that applies to multiple declarations within the scope of code that the annotation appears on. (Halloran)
<code>@Region</code>	Declares a new abstract region of state for the class to which this annotation is applied. (Greenhouse, Boyland)
<code>@RegionEffects</code>	Declares the regions that may be read or written during execution of the method or constructor to which this annotation is applied. (Greenhouse, Boyland)
<code>@RegionLock</code>	Declares that holding a particular lock, which may be a built-in (synchronization) lock, or may be an explicit <code>java.util.concurrent.Lock</code> , is required when a particular region of state is accessed. (Greenhouse)
<code>@RequiresLock</code>	Declares that the method or constructor to which this annotation applies assumes that the caller holds the named locks. (Greenhouse)
<code>@ReturnsLock</code>	Declares that the object returned by the method to which this annotation is applied is the named lock. (Greenhouse)
<code>@Revoke</code>	Declares that a particular thread role (<i>i.e.</i> , a named thread color) is revoked from a thread. (Sutherland)
<i><code>@Starts</code></i>	Declares what threads, if any, are started, <i>i.e.</i> , by <code>Thread.start()</code> , during the execution of the method or constructor to which this annotation is applied. (Halloran)
<code>@ThreadSafe</code>	The class to which this annotation is applied is thread-safe. (Goetz, <i>et al.</i> [51])
<code>@Transparent</code>	Declares that the annotated code may be legitimately be invoked from any thread. (Sutherland)
<i><code>@TypeSet</code></i>	Declares a named set of types to be used in <code>@MayReferTo</code> and <code>@Layer</code> annotations. (Halloran)
<code>@Unique</code>	Declares that the parameter, receiver, return value, or field to which this annotation is applied is a unique reference to an object. (Boyland)
<code>@Vis</code>	Declares that the annotated type, field, method, or constructor is part of the exported interface of the module it is contained within. (Sutherland)
<i><code>@Vouch</code></i>	Vouches for any “×” analysis result within the scope of code that the annotation appears on. (Halloran)

Figure 1.8: (Mb through Z) An alphabetical list of the annotations supported by the JSure prototype tool. Each annotation includes a brief description and the (parenthesized) name of its developer(s). Italicized annotations (also in blue) are contributions of this thesis. The majority of these annotations were developed by members of the Fluid project (with references to this prior work listed in Figure 1.3), however, the tool also supports the Goetz, *et al.* annotations from the book *Java Concurrency in Practice* [51]. The annotations are further described, with examples of their use, in Appendix A.

1.4.1 Supporting verification

A limitation of prior work by members of the Fluid project is that analysis result reporting is similar to traditional compiler error messages. If we consider our example, the “compiler-like” analysis results for the `BoundedFIFO` compilation unit would be reported as shown in Figure 1.9. We call this type of reporting “compiler-like” in the sense that the descriptive message output by the tool has to communicate the semantics of the finding. It differs from compiler output because the output of a consistent result is good news to the programmer, while compiler output is traditionally limited to error reporting (*i.e.*, no news is good news).

The analysis results report “points of consistency” and “points of inconsistency” found in the code. The analyses discussed above, and considered in our work, are *sound*, this means that for the analysis findings that are reports of potential inconsistency between stated promises and a program, there are no false negatives. That is, if there is an actual “point of inconsistency,” then there will necessarily be a corresponding finding. On the other hand, there may be false positives, in the sense that the analysis may fail to find evidence of actual consistency. Sound analyses are also referred to as *conservative* in the traditional compiler literature [2].

In this section we discuss the following three limitations of analysis result reporting done in a manner similar to traditional compiler error messages:

- (1) Relationships between promises are lost or only hinted at in the textual description of the reported result.
- (2) Due to (1), the impact of even a single “point of inconsistency” on the consistency of other promises is difficult to understand.
- (3) The reported results do not directly answer the programmer’s verification question, “Is my annotation consistent with the code?”

Following this discussion we introduce the *drop-sea* proof management system and describe the features that it provides to help overcome these limitations.

Lost relationships between promises

Some relationships between the promises in `BoundedFIFO` are not evident from the results reported in Figure 1.9. In this style of reporting the analysis can only “hint” at these relationships through the textual description reported to explain the result to the analysis tool user. We consider this limitation in the context of the following three examples from the findings listed in Figure 1.9:

- Analysis result f_4 is a consistent finding about the write to the field `size` that occurs at line 13. Notice, however, that at this point in the program `FIFOLock` (*i.e.*, `this`) is not held. As we discussed above, object construction is a special case, and indeed the lock analysis “trusts” that the `@Unique("return")` promise at line 10 on the `BoundedFIFO` constructor is consistent with the code.
- Analysis result f_{24} is a consistent finding about the read of the field `numElts` that occurs at line 39 in the implementation of the `length()` method. Notice, however, that

Lock Policy Analysis Results for BoundedFIFO

	Finding	About	Description
f_1	+	r_1	thread-confined access to <code>numElts</code> at line 8
f_2	+	r_1	thread-confined access to <code>first</code> at line 8
f_3	+	r_1	thread-confined access to <code>next</code> at line 8
f_4	+	r_1	thread-confined access to <code>size</code> at line 13
f_5	+	r_1	thread-confined access to <code>buf</code> at line 14
f_6	+	r_1	FIFOLock held for access to <code>numElts</code> at line 19
f_7	+	r_1	FIFOLock held for access to <code>buf</code> at line 20
f_8	+	r_1	FIFOLock held for access to <code>first</code> at line 20
f_9	+	r_1	FIFOLock held for access to <code>buf[first]</code> at line 20
f_{10}	+	r_1	FIFOLock held for access to <code>first</code> at line 21
f_{11}	+	r_1	FIFOLock held for access to <code>size</code> at line 21
f_{12}	+	r_1	FIFOLock held for access to <code>first</code> at line 21
f_{13}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 22
f_{14}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 28
f_{15}	+	r_1	FIFOLock held for access to <code>size</code> at line 28
f_{16}	+	r_1	FIFOLock held for access to <code>buf</code> at line 29
f_{17}	+	r_1	FIFOLock held for access to <code>next</code> at line 29
f_{18}	+	r_1	FIFOLock held for access to <code>buf[next]</code> at line 29
f_{19}	+	r_1	FIFOLock held for access to <code>next</code> at line 30
f_{20}	+	r_1	FIFOLock held for access to <code>size</code> at line 30
f_{21}	+	r_1	FIFOLock held for access to <code>next</code> at line 30
f_{22}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 31
f_{23}	+	r_1	FIFOLock held for access to <code>size</code> at line 36
f_{24}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 39
f_{25}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 42
f_{26}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 45
f_{27}	+	r_1	FIFOLock held for access to <code>size</code> at line 45
f_{28}	+	r_1	FIFOLock held for access to <code>numElts</code> at line 48
f_{29}	+	r_1	FIFOLock held for access to <code>size</code> at line 48

Uniqueness Analysis Results for BoundedFIFO

	Finding	About	Description
f_{30}	+	r_4	reference held by <code>buf</code> is unique (<i>i.e.</i> , unaliased)
f_{31}	+	r_{10}	constructor does not alias <code>this</code>
f_{32}	+	r_{10}	<code>super()</code> promises not to alias <code>this</code>

Figure 1.9: “Compiler-like” analysis results for BoundedFIFO compilation unit. (Top) The analysis results reported from the Greenhouse lock policy analysis [53]. (Bottom) The analysis results reported from the Boyland uniqueness analysis [20, 21]. Each analysis result is labeled (*e.g.*, f_1) for identification. A finding of “+” indicates a “point of consistency” between the code and the promise the result is about (no “points of inconsistency” were found in this code). The promise the result is about is referred to by an r with a subscript that indicates the line of code where the promise appears in Figure 1.5 (*e.g.*, r_1 refers to the `@RegionLock` promise at line 1 in Figure 1.5). A short description is reported to explain the result to the analysis tool user.

within this method `FIFOLock` (*i.e.*, `this`) is not acquired. This result “trusts” that the `@RequiresLock("FIFOLock")` promise about the `length()` method is consistent with any code that invokes the method.

- Analysis result f_{32} is a consistent finding that the implicit call to the superclass constructor (*i.e.*, the no-argument constructor for `Object`) respects the `@Unique("return")` promise at line 10 that the receiver will not be aliased. The no-argument constructor for `Object` promises `@Unique("return")` as a “standoff annotation” using XML structures because `Object` is part of the Java standard library and is typically used in binary form. This promise on the no-argument constructor for `Object` is “trusted” by this analysis result.

None of the relationships between promises described in the above examples is clearly communicated by the textual description for that particular result.

Unknown impact of an “x” result

Because the analyses we employ are sound, if all reported analysis results are consistent then the promises that those results are about are verified—they express an invariant on all possible executions of the code examined. However, if any “x” results are reported does that mean that all the promises annotated in the code base are unverifiable? Maybe, maybe not. Using this style of analysis result reporting it can be difficult for the tool user to understand the impact of an “x” result on the consistency of promises with the code. Clearly, this problem is related to the previous one, if we do not understand the relationships among promises we cannot reason about the impact of any particular “x” result on the consistency of related promises.

As an example of this problem, consider the code and analysis results for the `Dispatcher` class shown in Figure 1.10. The three reported analysis results are “x” results because the code does not respect the concurrency policy expressed by the promises in `BoundedFIFO`—it synchronizes on the wrong lock: `this` which is the `Dispatcher` instance, rather than `fifo` which is the `BoundedFIFO` instance. These three “points of inconsistency” make the `@RequiresLock("FIFOLock")` promises on the `length()`, `get()`, and `wasFull()` methods unverifiable, and correspondingly the `@RegionLock` promise at line 1. This impact is not evident to the programmer from the analysis output.

Not answering the right question

A fundamental limitation of the compiler-like style of analysis results reporting is that it doesn’t directly answer the question that the programmer is most concerned with, “Are my annotations consistent with my code?” An examination of the analysis results about a particular promise may not give the programmer the correct answer to this question. For example, by examination of the analysis results about the `@RegionLock` promise at line 1 of `BoundedFIFO` (referred to as r_1) reported in Figure 1.9 and Figure 1.10, the programmer could conclude that this promise is verified. However, the three “x” results reported in Figure 1.10 about `@RequiresLock` promises in the `BoundedFIFO` code indirectly make this conclusion invalid.

```

50 public class Dispatcher {
51
52     private final BoundedFIFO fifo;
53
54     Dispatcher(BoundedFIFO value) {
55         fifo = value;
56     }
57
58     LoggingEvent get() {
59         synchronized (this) { // Broken - acquires the wrong lock
60             LoggingEvent e;
61             while (fifo.length() == 0) {
62                 try {
63                     fifo.wait();
64                 } catch (InterruptedException ignore) { }
65             }
66             e = fifo.get();
67             if (fifo.wasFull()) {
68                 fifo.notify();
69             }
70             return e;
71         }
72     }
73     ...
74 }

```

Lock Policy Analysis Results for Dispatcher

	Finding	About	Description
f_{33}	×	r_{38}	FIFOLock not held when invoking <code>length()</code> at line 61
f_{34}	×	r_{17}	FIFOLock not held when invoking <code>get()</code> at line 66
f_{35}	×	r_{44}	FIFOLock not held when invoking <code>wasFull()</code> at line 67

Figure 1.10: Code and “compiler-like” analysis results for the `Dispatcher` class used by the dispatcher thread to read events from the `BoundedFIFO` instance and handle them (*e.g.*, writes them out to a disk file). (Top) Elided Java code for the `Dispatcher` class. This class holds the wrong lock, `this` rather than `fifo`, when invoking methods on the shared `BoundedFIFO` instance. (Bottom) The analysis results reported from the Greenhouse lock policy analysis [53]. A finding of “×” indicates a “point of inconsistency” between the code and the promise the result is about (no inconsistencies were found in this code). Line and result numbering is continued starting after the last line of the `BoundedFIFO` code to keep references unambiguous.

To be safe, the programmer has to attempt to fix any reported “x” analysis result so that he or she can be sure that their promises are consistent with the code.

Overcoming these limitations: The drop-sea proof management system

In Chapter 2 we develop a formal model that describes the construction of proofs within our approach to analysis-based verification. This includes specifying precisely how “plug-in” program analyses report their findings and the development of an automatable proof calculus to create program- or component-level results based upon these findings. This proof calculus allows separate analysis of components and composition of the results such that the outcome corresponds to that of a whole-program analysis. To support automated reasoning in the JSure prototype tool we introduce the *drop-sea* proof management system (Chapter 3). By *proof management* we refer to the manipulation of formal proofs and proof fragments, *i.e.*, lemmas, as data structures. Drop-sea manages the results reported by “plug-in” program analyses and automates the proof calculus developed in Chapter 2 to produce verification results based upon these findings.

We now discuss how drop-sea helps to overcome the three limitations of the “compiler-like” analysis result reporting described above.

- **Overcoming – Lost relationships between promises.** Our approach makes these lost relationships between promises explicit by taking advantage of the observation that an understanding about how particular promises relate to one another is embodied in the constituent analyses. Our approach provides a mechanism for its elicitation and use. If an analysis result “trusts” a promise then we refer to that promise as a *prerequisite assertion*. Each analysis result is allowed to report a prerequisite assertion. The analysis results for f_4 , f_{24} , and f_{32} used in the three “lost relationship” examples above, now including their prerequisite assertions, are shown in Figure 1.11.

Prerequisite assertions, as presented in Chapter 2, are allowed to be formulas in *promise logic*: an intuitionistic (or constructivist) sequent calculus is used for proving sequents in promise logic. In addition, analyses propose promises that may or may not exist in the code as their prerequisite assertions. A special analysis called *promise matching* is used to match each proposed promise with a real promise in the code. The prerequisite assertions shown in Figure 1.11, to simplify this example, may be considered to be post promise matching⁸.

In our approach, constituent analyses report their results to drop-sea. These results are modeled as a graph. The example drop-sea structures presented in this chapter are all trees, however, in the presence of recursive calls in the program the resulting structure is a graph.

A portion of the drop-sea graph for the `BoundedFIFO` and `Dispatcher` classes, focused on the analysis results for f_4 , f_{24} , and f_{32} , after the lock policy and uniqueness analyses have reported their results is shown in Figure 1.12. Drop-sea is named to invoke the metaphor of drops of water in a sea. Drop-sea is implemented as an interconnected

⁸The prerequisite assertion for f_4 shown in Figure 1.11 is further simplified to remove the possibility that thread-confined access to the field is assured with effects rather than uniqueness as this would introduce a disjunction into its prerequisite assertion—disjunction is supported by the proof system introduced in Chapter 2.

Analysis Results for BoundedFIFO

	Finding	About	Prerequisite	Description
f_4	+	r_1	r_{10}	thread-confined access to size at line 13
...				
f_{24}	+	r_1	r_{38}	FIFOLock held for access to numElts at line 39
...				
f_{32}	+	r_{10}	r_{78}	super() promises not to alias this

```

75 <package name="java.lang">
76   <class name="Object">
77     <constructor>
78       <Unique>return</Unique>
79       ...
80     </constructor>
81   </class>
82 </package>

```

Analysis Results for java.lang.Object

	Finding	About	Prerequisite	Description
f_{36}	+	r_{78}	\top	constructor does not alias this
...				

Figure 1.11: (Top) Elided analysis results for **BoundedFIFO** reporting an explicit prerequisite assertion for each “point of consistency” found in the code by the constituent analysis. (Middle) Elided promises about the no-argument constructor of the **java.lang.Object** class (the superclass of **BoundedFIFO**). The **@Unique("return")** promise is made as a “standoff annotation” using XML structures because **Object** is part of the Java standard library and is typically used in binary form. Annotation via XML is equivalent to direct annotation of code. This promise, referred to as r_{78} , is the prerequisite assertion for the analysis result f_{32} . (Bottom) Elided analysis results for **java.lang.Object**. The result about the **@Unique("return")** promise on the no-argument constructor does not require a prerequisite assertion, the symbol \top is used to express this lack of a prerequisite. (As is presented in the next chapter, the prerequisite assertion is logical formula that constrains the verification of the promise the result is about, \top , which represents the tautology, indicates no constraint.) Line and result numbering is continued starting after the last line of the **Dispatcher** code to keep references unambiguous.

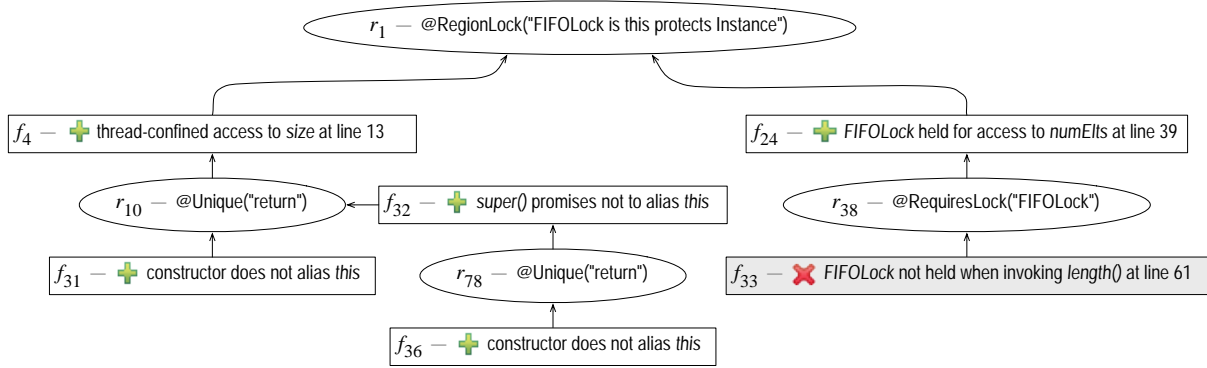


Figure 1.12: A portion of the drop-sea graph for the **BoundedFIFO** and **Dispatcher** classes after the lock policy and uniqueness analyses have reported their results for the promises and code in Figure 1.5 and Figure 1.10. Promise drops, which are independent of any specific analysis, are represented by ovals. Result drops, which are reported from constituent analyses, are represented as rectangles. Each result drop shows a “+” for a conservative judgment by the analysis of model-code consistency, or an “✗” otherwise. A directed edge from a promise drop to a result drop indicates that the promise is a prerequisite assertion for that result. A directed edge from a result drop to a promise drop indicates that the result is about that promise.

graph of dependencies between “drops” (the nodes) that represent promises, analysis results, and supporting information in a container we call the “sea”. The graph represents a chain of evidence about each promise, with regard to its consistency as well as the what that consistency depends upon. By explicitly tracking dependencies, drop-sea performs the role of truth maintenance for the system.

- **Overcoming – Unknown impact of an “✗” result.** The drop-sea graph shown in Figure 1.12 can be analyzed to determine the verification result for each promise. This analysis of the graph automates the proof calculus that we developed to create program- or component-level verification results. The “decorated” graph after computing verification results on the graph in Figure 1.12 is shown in Figure 1.13. This graph makes it clear that the “✗” analysis result, f_{33} , makes it impossible for the tool to verify the @RequiresLock promise at line 38 and the @RegionLock promise at line 1.

The verification results computed on the drop-sea graph are only meaningful on promises. However, by keeping “results” on the other types of nodes we can help the user to understand why a promise can’t be verified. The user can follow the trail of “✗”s to determine which “✗” result or set of “✗” results caused the verification to fail. This trail of “✗” results is visible in the JSure tool user interface as shown in Figure 1.14.

The JSure tool uses a tree to display the drop-sea graph with computed verification results to the tool user. The tool view in Figure 1.14 is expanded to allow a comparison of how the tool displays the drop-sea graph nodes shown in Figure 1.13. As in the drop-sea graph, verification results, indicated by small icons (referred to in Eclipse as decorators) to the lower-left, are really only meaningful on promises, however, we decorate the tree to help the tool user track down “✗” results causing a promise to be unverifiable.

- **Overcoming – Not answering the right question** The addition of drop-sea and the computation of verification results allows the JSure tool to answer the question

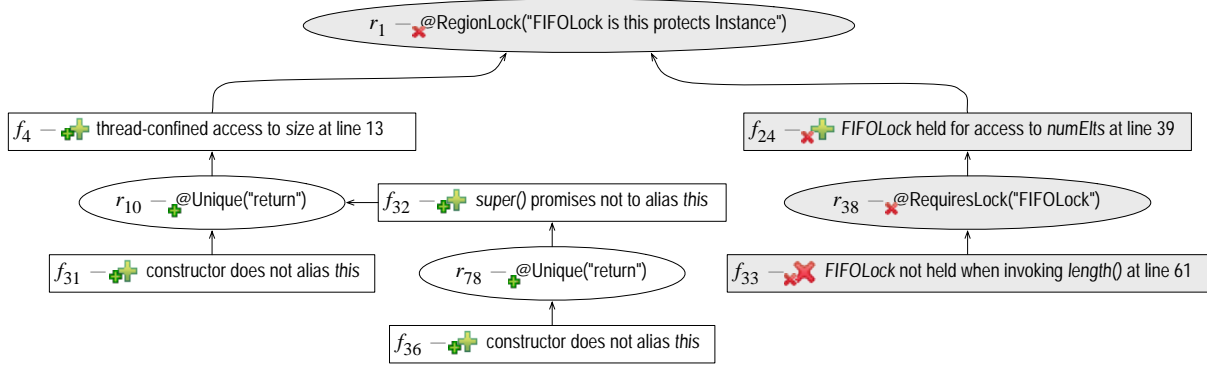


Figure 1.13: A portion of the drop-sea graph for the `BoundedFIFO` and `Dispatcher` classes showing computed verification results. A small “+” (to the lower-left) indicates model-code consistency. A small “x” (to the lower-left) indicates a failure to prove model-code consistency (the grey nodes).

that the programmer is most concerned with, “Are my annotations consistent with my code?” The answer reported about the `@RegionLock` promise at line 1 of `BoundedFIFO` is no.

Tool interaction toward consistency

The programmer would likely not be satisfied with leaving the promises in `BoundedFIFO` in the state of partial consistency reported by the JSure tool in Figure 1.14. We now discuss the tool interaction with the programmer to work toward model-code consistency. This interaction is illustrated in Figure 1.15.

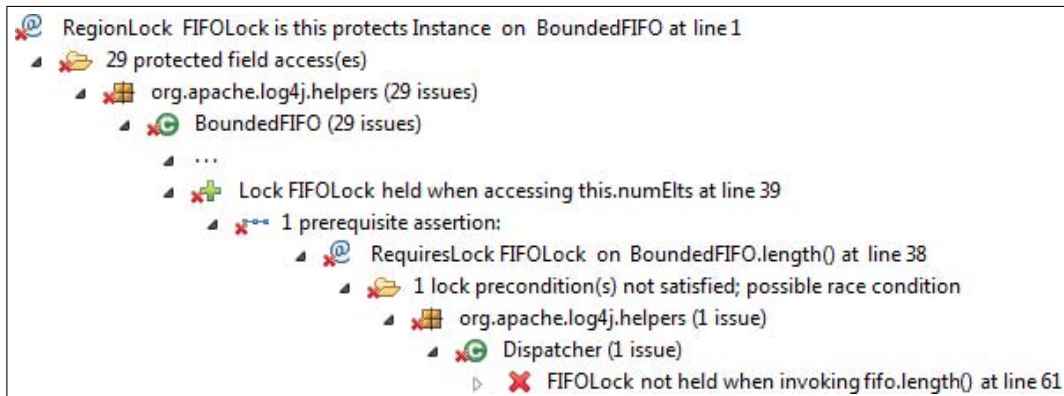
- The programmer would, from the unverifiable promise of interest, follow the trail of “x”s to find a result that caused the verification to fail. In Figure 1.15 the programmer is interested in verifying the `@RegionLock` promise and follows the trail of “x”s to the call to the `length()` method at line 61 of the `Dispatcher` class (shown in Figure 1.10).
- A double-click on the “x” analysis result highlights the identified code in the Java editor. In this case, the code identified by the “x” analysis result is the unprotected call to the `length()` method. Upon examination of the code the programmer can determine that the code is errant, it uses the wrong lock, and change line 59 to synchronize on `fifo` rather than `this`. In this example the code is incorrect, however, it can be the case that the expressed model is incorrect and the code is correct. In this situation the programmer would fix, or add to, the model rather than changing the code.
- After saving the change to line 59, the JSure tool re-runs its analysis and determines that the `@RegionLock` promise is consistent with the code. The programmer may continue to express more models and verify them with the tool or choose to move on to other work.

1.4.2 Supporting model expression

A possible criticism of our approach to analysis-based verification is the number of annotations required to allow the constituent analyses to verify a particular model. For our `BoundedFIFO`



Figure 1.14: A JSure tool view showing a portion of the verification results for the `BoundedFIFO` and `Dispatcher` classes. The results show the 29 accesses to protected state reported in Figure 1.9. The tool view is expanded to show how it displays the drop-sea graph nodes shown in Figure 1.13. A small “x” (to the lower-left) indicates a failure to prove model-code consistency.



Double-clicking on the analysis result (at the bottom) selects the unprotected call in the source code of Dispatcher.java

```

58 LoggingEvent get() {
59     synchronized (this) {
60         LoggingEvent e;
61         while (fifo.length() == 0) {
62             try {
63                 fifo.wait();
64             } catch (InterruptedException ignore) { }
65         }
66         e = fifo.get();
67         if (fifo.wasFull()) {
68             fifo.notify();
69         }
70         return e;
71     }
72 }
  
```

The programmer determines that the code is wrong and fixes line 59

```

58 LoggingEvent get() {
59     synchronized (fifo) {
60         LoggingEvent e;
61         while (fifo.length() == 0) {
  
```

The JSure tool re-runs its analysis and determines that the promise is consistent with the code

@RegionLock FIFOlock is this protects Instance on BoundedFIFO at line 1

Figure 1.15: Programmer-tool interaction to fix the errant Dispatcher code (in Figure 1.10) and verify the @RegionLock promise at line 1 in BoundedFIFO (as shown in Figure 1.5).

example we had to enter 11 promises to express a model complete enough for the JSure tool to verify. In this section we introduce two approaches to assist the programmer with model expression: *proposed promises* and the *scoped promise*, `@Promise`. These two techniques help the programmer avoid tedious annotation of their code. In this section we continue to use the `BoundedFIFO` example to describe both of these techniques.

Proposed promises: Helping the tool user complete partial models

Our approach allows constituent analyses to report any necessary prerequisite assertions as part of each analysis result. Analyses, when they report a prerequisite assertion, propose promises that may or may not exist in the code. A special analysis called *promise matching* is used to “match” each proposed promise with a programmer-expressed promise in the code. A match indicates that the assertion represented by the programmer-expressed promise in the code implies the assertion represented by the proposed promise. In this section we are concerned about what the tool does if no “match” can be found, *i.e.*, a promise proposed by a constituent analysis is not implied by any promises in the code base.

The computation that produces verification results is able to use the remaining proposed promises (after promise matching) to determine the “weakest” prerequisite assertion for each promise in the code base. The calculation is similar to the computation of the weakest precondition for a block of code in the classic verification literature [40], however, a prerequisite assertion is not the same as a precondition. Traditionally, a precondition, ψ , would be at a program point before a block of code, B , and a postcondition, ϕ , would be after it; we would say that if B starts execution in a state that satisfies ψ , then the state after running B will satisfy ϕ [79]. In our approach to analysis-based verification, the corresponding terms, prerequisite assertion and consequential assertion (*i.e.*, the promise we are verifying), are used with respect to the consistency of promises with the program. Specifically, consistency of a *prerequisite* assertion ψ with the program is a sufficient condition to establish the consistency of the *consequential* assertion with the program.

Our ability to compute the weakest prerequisite assertion for a particular promise allows the tool to propose annotations to the code base that can be reviewed and accepted by the tool user. Using the `BoundedFIFO` example we now discuss the tool interaction with the programmer to help to annotate the `BoundedFIFO` code from a single promise.

It can be argued that the annotations to `BoundedFIFO` shown in Figure 1.5, except for the `@RegionLock` annotation at line 1 that expresses the programmer’s intended locking policy for the shared state of the class, are rather bureaucratic and tedious for the user to manually enter—or to even know that they need to be entered. The textual description of an analysis result is a poor way to communicate a “need” for further annotation to the tool user because a precise expression of both the location and content of the needed annotation can become lengthy and potentially confusing. Therefore, the JSure tool proposes “missing” promises to the user that it can, after approval by the user, place within the code.



The tool interaction to help annotate the `BoundedFIFO` code from a single programmer-entered promise is illustrated in Figure 1.16. This example assumes that the `Dispatcher` code has been fixed (we discuss the behavior if it has not been fixed below).

- The programmer expresses the locking policy for shared state defined in the `BoundedFIFO` class with the promise `@RegionLock("FIFOLock is this protects Instance")`.











The programmer enters the @RegionLock promise into BoundedFIFO

```
1 @RegionLock("FIFOLock is this protects Instance")
2 public class BoundedFIFO {
3
4     LoggingEvent[] buf;
```

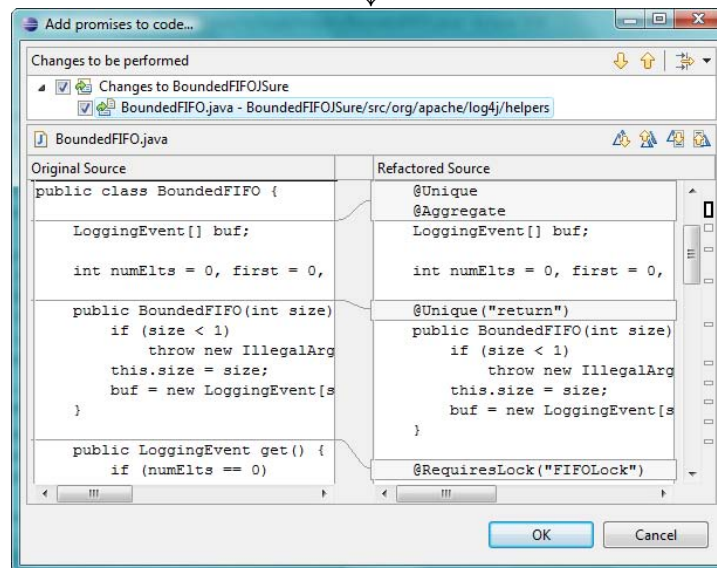
JSure cannot verify the promise, but it proposes “missing” promises to the programmer

 RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1
 27 unprotected field accesses; possible race condition detected

Proposed Promises (BoundedFIFOJSure)

Description	Resource	Line
 Aggregate	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	8
 Unique	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	8
 Unique("return")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	12
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	19
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	29
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	38
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	42
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	46
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	50
 RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	54

Using the (circled) context menu, the programmer directs the tool to add the promises



With the 10 additional promises in the code, JSure can verify the locking model


 RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1

Figure 1.16: Programmer-tool interaction to use proposed promises to help annotate BoundedFIFO (as shown in Figure 1.5) from one programmer-entered @RegionLock annotation.

- The JSure tool reports that the `@RegionLock` promise cannot be verified. The tool reports that the 27 accesses to the protected fields in the class are performed in code that the analysis cannot verify is holding the correct lock. (Verifying analysis are modular and do not check the whole program.)
- The programmer could begin to follow the trail of “x”s to examine the results that caused the verification to fail, but the tool has proposed 10 promises that it has determined, through the computation of the weakest prerequisite assertion, may be missing from the programmer’s code.
- The programmer selects all these promises and asks the tool, via a context menu, to add them to the code base.
- The JSure tool previews the edit to the code for the programmer to confirm.
- The programmer confirms the edit and, with the 10 additional promises now in the code, JSure re-runs its analysis and determines that the `@RegionLock` promise is consistent with the code.

The use of proposed promises to help the user with model expression is effective, in part, because the perplexity, the number of potential models that can make a particular promise verifiable, is low. This property holds for the analyses supported by the JSure tool today (shown in Figure 1.3), but it may not hold for all useful analyses⁹.

In the presence of errant code, such as the implementation of the `Dispatcher` class shown in Figure 1.10, the tool can propose a promise for annotation into the code that may seem “odd” to the programmer with respect to its location or, after it is added, its inability to be verified. For example, because the `get()` method in the `Dispatcher` class acquires the wrong lock (at line 59) the weakest prerequisite assertion would include a proposal to add a `@RequiresLock("FIFOLock")` to the `get()` method in the `Dispatcher` class. The location of this promise, *i.e.*, outside of the `BoundedFIFO` class, should prompt the programmer to consider that something may be wrong with the code. If, however, the promise is added to the code by the programmer, the JSure tool will fail to verify it—prompting further investigation by the programmer into the errant implementation of the `Dispatcher` class.

@Promise: Helping the tool user avoid repetitive annotation

Scoped promises are promises that act on other promises or analysis results within a static scope of code. The `@Promise` scoped promise can be used to help the tool user avoid repetitive annotation. The use of `@Promise` allows the user to declare—using an aspect-like syntax [69]—the declarations (*e.g.*, types, methods, and fields) within its static scope of where a “payload” promise is to be placed.

Figure 1.17 shows the use of `@Promise` to place payload promises onto declarations within the `BoundedFIFO` class. The pattern `new(**)`, used in the promise at line 3, matches all constructors declared within the class with any number of parameters (including zero)—this pattern will match the one constructor in the class and place the payload promise

⁹One analysis where this property may not hold is the permissions analysis developed by Boyland, Retert, and Zhao [23, 22]. This analysis, in some cases, can result in a weakest prerequisite assertion that consists of the disjunction of hundreds of assertions.

```

1 @RegionLock("FIFOLock is this protects Instance")
2 @Promises({
3   @Promise("@Unique(return)          for new(**)"),
4   @Promise("@RequiresLock(FIFOLock) for *(**)")
5 })
6 public class BoundedFIFO {
7
8   @Unique
9   @Aggregate
10  LoggingEvent[] buf;
11  ...
12 }

```

Figure 1.17: Elided Java code for the `BoundedFIFO` class using two `@Promise` annotations to place annotations onto declarations within the class. The first annotation (at line 3) places a `@Unique("return")` annotation all the class’s constructors. The second annotation (at line 4) places a `@RequiresLock("FIFOLock")` annotation on all the class’s methods. The use of `@Promise` reduces the number of annotations from the 11 shown in Figure 1.5 down to 6. The `@Promises` annotation allows us to place more than one `@Promise` annotation on the class.

`@Unique("return")` on it. The pattern `*(**)`, used in the promise at line 4, matches all methods (including static methods¹⁰) declared within the class with any number of parameters—this pattern will match the 7 methods declared in the class and place the payload promise `@RequiresLock("FIFOLock")` on all of them. The use of `@Promise` annotations allows the programmer to convey that *all* declarations should promise something—those that exist in the code today *and those added to the code in the future*. In general, `@Promise` allows our approach to support an “enter-once” principle. If the programmer is expressing a unitary concept, then its expression should not have to be scattered throughout the code base.

We refer to promises that are “created” by `@Promise` as *virtual promises*. Programs on operating systems with virtual memory are unaware that they do not directly address physical memory. Similarly, verifying analyses within an analysis-based verification system do not realize that virtual promises are not directly annotated in the code. In our `BoundedFIFO` example, no verifying analysis would note a difference between the `@Unique("return")` and `@RequiresLock("FIFOLock")` promises in Figure 1.5 and the virtual promises created by the `@Promise` annotations at lines 3 and 4 in Figure 1.17. This abstraction, performed by the infrastructure of the analysis-based verification tool, simplifies construction of new program analyses.

The aspect-like syntax can be avoided if the intent is to place the payload promise on all declarations where it is meaningful within a static scope of code. Figure 1.18 shows an example where `@Promise` is used to place a payload promise of `@InLayer("MODEL")` on all types declared within the `edu.ait.smallworld.model` package. The `for` clause, used in the examples in Figure 1.17, is omitted from the `@Promise` syntax in this case.

Sutherland used `@Promise` to reduce the number of programmer-expressed annotations in a field trial on the 140 KSLOC Electric open-source VLSI-design tool. He reports in [103]:

“With [`@Promise`], we were able to avoid writing nearly two thousand annotations, and instead write only fifty-four—six scoped promises in each of nine packages.”

¹⁰To match all only non-static methods the pattern `!static *(**)` would be used.

```
@Promise("@InLayer(MODEL)")
package edu.afit.smallworld.model;
```

Figure 1.18: The use of an `@Promise` annotation in a `package-info.java` file to place an `@InLayer` promise on every type declared within the `edu.afit.smallworld.model` package. The `@InLayer` promise, introduced in Chapter 6, is used to map types into static layer—part of a structural model of the code that can be verified by JSure.

```
@Promises({
    @Promise("@Color(DBExaminer | DBChanger) for get*(**) | is*(**) | same*(**)",
    @Promise("@Color(DBExaminer | DBChanger) for compare*(**) | connectsTo*(**)",
    @Promise("@Color(DBExaminer | DBChanger) for contains*(**) | describe()",
    @Promise("@Color(DBExaminer | DBChanger) for find*(**) | num*(**)",
    @Promise("@Color(DBChanger) for set*(**) | make*(**) | modify*(**)",
    @Promise("@Color(DBChanger) for clear*() | new*(**) | add*(**)"
})
package com.sun.electric.database.network;
```

Figure 1.19: A `package-info.java` file within the Electric open-source VLSI-design tool annotated with six `@Promise` annotations by Sutherland to place `@Color` promises as part of a field trial of the JSure tool and his thread coloring analysis [105].

An example of the annotations made to one of the nine Electric packages is shown in Figure 1.19. The details of the pattern matching syntax used in this example are presented in Chapter 3.

1.4.3 Supporting contingencies

Our approach allows three types of unverified contingencies to exist in the chain of evidence about a promise:

1. Using the `@Vouch` promise, a programmer can vouch for an overly conservative analysis result—in effect, changing it from an “x” to a “+”.
2. Using the `@Assume` promise, a programmer can assume an assertion holds about a component that is outside of the programmer’s scope of interest, *e.g.*, on the other side of an organizational or contractual boundary.
3. The programmer can turn off, using the JSure tool preferences, a particular program analysis. This action causes all the promises checked by that analysis to have no results. The tool “trusts” that these promises are consistent.

`@Vouch` and `@Assume` perform differing roles in our system—`@Vouch` acts on (“x”) *analysis results*, whereas `@Assume` acts on (absent) *promises*.

Taking one or more of the above actions introduces a contingency into any proof that relies upon them. Drop-sea explicitly tracks these contingencies and flags them with a *red dot*. The red dot, so named because of its dark red “●” icon in the tool user interface, indicates

that a programmer is willing to prick a finger and vouch for the unverified contingency with a small drop of blood—at least virtually.

@Vouch – Dealing with overly conservative analysis results

An example of a programmer vouching for unverifiable code is shown in Figure 1.20. The example is from the Apache Hadoop MapReduce project¹¹. Hadoop MapReduce is a programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes. In this example several programmer vouches (Figure 1.20 shows one of these in the `CapacityTestUtils` code snippet) have been used to indicate that test code violates a locking policy that protects the status of a job in progress. In the tool output the red dot icon is shown to the upper-left of any result that depends upon a the use of `@Vouch`. Similar to computed verification results this indication of a contingency is only meaningful on promises, however, we decorate the tree to help the tool user track down which contingencies caused the red dot to appear on a promise.

While pragmatic, we admit that `@Vouch` exhibits the same potential for abuse that we expressed concern about regarding the `@SupressWarnings` annotation in Section 1.3.2, namely that the code base can become littered with annotations having tool-specific semantics—their meaning tied to the implementation of a particular analysis for the purpose of stopping its complaints about what is, from the programmer’s point of view, correct code. However, given the fundamental limitations of static analysis, annotations like `@Vouch` for analysis-based verification and `@SupressWarnings` for heuristics-based static analysis (and the Java compiler itself) are necessary, though we advocate trying to limit their use as much as possible.

@Assume – Expressing expectations about other components

We allow the programmer to assume, within a limited scope of code, that a promise exists about a particular declaration outside of that scope of code. This type of assumption is expressed using the `@Assume` scoped promise. The `@Assume` scoped promise allows the user to declare—using an aspect-like syntax [69]—declarations (e.g., types, methods, and fields) outside of its static scope that the user would like the “payload” promise to be consistent for. Virtual promises created by `@Assume` are not checked by verifying analyses. In addition, they are only visible to a verifying analysis when it is examining the compilation unit where the `@Assume` annotation appears.

Figure 1.21 shows an example where a local assumption is introduced into the program. This example, again drawn from the Apache Hadoop MapReduce project, provides another example of the tool-generated promises using the proposed promise feature.

- The programmer expresses that the `getDescendantContainerQueues()` method of the `JobQueue` class should start no threads by placing a `@Starts("nothing")` promise on the method.
- The JSure tool reports that the `@Starts("nothing")` promise cannot be verified because the no-argument constructor for `java.util.ArrayList` does not promise to start no threads. The tool, however, has proposed that this promise be added.

¹¹<http://hadoop.apache.org/mapreduce/>

```

@Region("StatusState")
@RegionLock("StatusLock is this protects StatusState")
public class JobInProgress {
    @InRegion("StatusState")
    JobStatus status;
    ...
}

```

```

public class CapacityTestUtils {
    @Vouch("This code is used only for testing")
    static class FakeJobInProgress extends JobInProgress { ... }
    ...
}

```

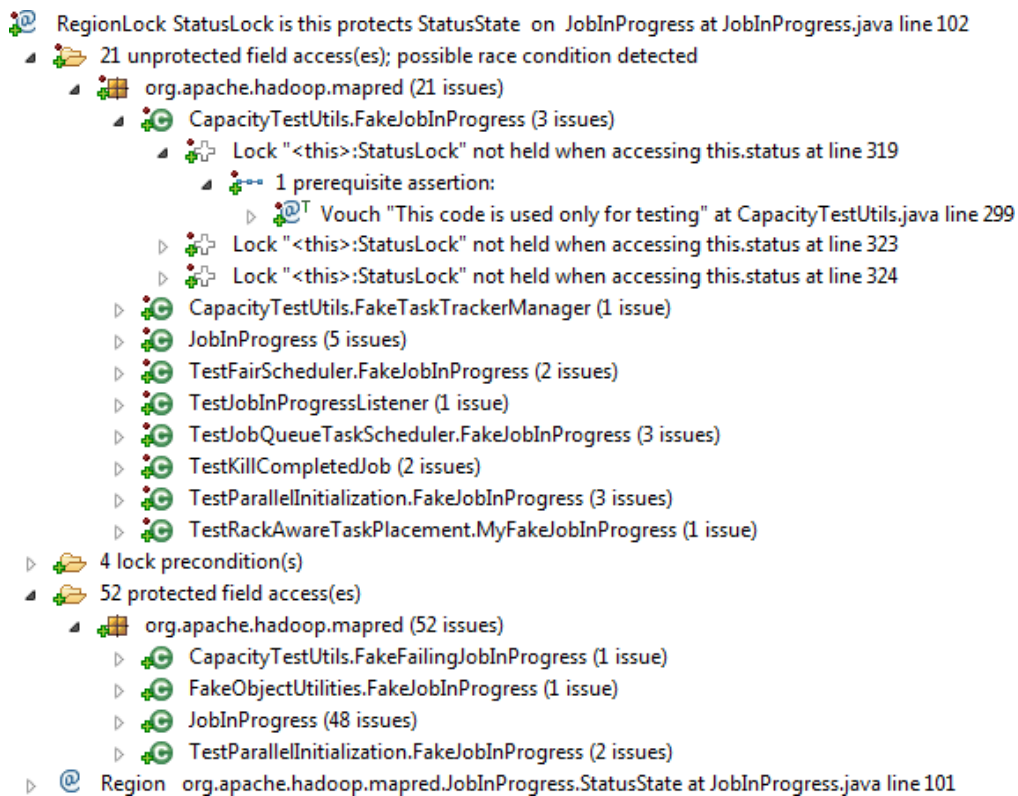


Figure 1.20: An example of using the `@Vouch` promise to indicate that test code is intentionally unverifiable. (Top) Elided Java code for the `JobInProgress` class from Hadoop’s *MapReduce* project including a locking model, `StatusLock`, that declares that a lock on the receiver (*i.e.*, `this`) is used to protect reads and writes to the field `status`. The *MapReduce* project contains 13 subclasses of `JobInProgress` that access this field. (Middle) Elided Java code for the `CapacityTestUtils` class that declares, as a nested class, a subclass of `JobInProgress` called `FakeJobInProgress`. The `@Vouch` annotation states this class is unverifiable because it is test code. (Bottom) JSure screenshot of the results for the verification of the `@RegionLock` promise on `JobInProgress`. The icon for any “×” analysis results that are within the scope of the `@Vouch` are changed to a (hollow grey) “+” with the `@Vouch` as a prerequisite assertion. The `@Vouch` promise in the results is identified by a T decorator to the upper-right of the @ icon, indicating that it is trusted. Because the programmer’s vouch is not verified by analysis, a red dot is introduced above any verification result that depends upon it. The red dot highlights a contingency to the tool user.

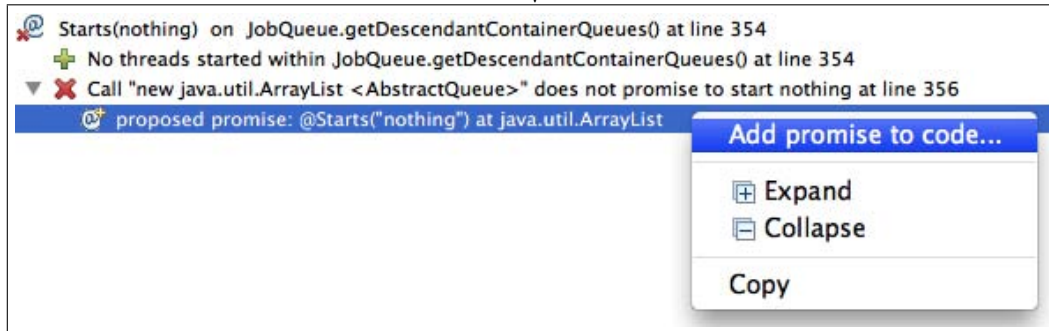
The programmer specifies that a method in the `JobQueue` class should start no threads

```

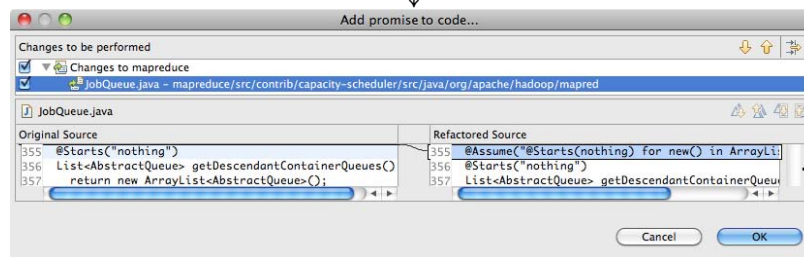
354 @Starts("nothing")
355 List<AbstractQueue> getDescendantContainerQueues() {
356     return new ArrayList<AbstractQueue>();
357 }
358
359 public void jobUpdated(JobChangeEvent event) {

```

JSure cannot verify the promise, but it proposes a “missing” promise to the programmer



Using the context menu, the programmer directs the tool to add the promise—because the promise is on a library class, *i.e.*, `java.util.ArrayList`, it is added as a local assumption



```

355 @Assume("@Starts(nothing) for new() in ArrayList in java.util")
356 @Starts("nothing")
357 List<AbstractQueue> getDescendantContainerQueues() {
358     return new ArrayList<AbstractQueue>();
359 }

```

With the assumed promise in the code, JSure can verify the model

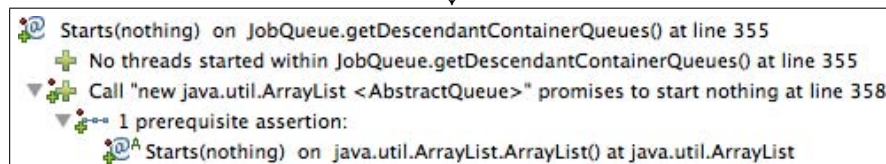


Figure 1.21: Programmer–tool interaction of using proposed promises to add a local assumption, `@Assume`, to the `JobQueue` class in Hadoop’s *MapReduce* project. Assumed promises are identified in the tool user interface by an `A` decorator to the upper-right of the `@` icon. Because the assumption is not verified by analysis, a red dot is introduced above any verification result that depends upon it.

- The programmer selects this promise and asks the tool, via the context menu, to add it to the code.
- The `java.util.ArrayList` class is included in the Hadoop MapReduce project as a library (*i.e.*, a JAR file), therefore, the JSure tool proposes that the method that needs this design intent annotate it as a local assumption with the promise:

```
@Assume("@Starts(nothing) for new() in ArrayList in java.util")
```

This promise assumes that there is a consistent `@Starts("nothing")` promise on the no-argument constructor, specified by the pattern “`new()`”, for the `java.util.ArrayList` class, specified by the pattern “`in ArrayList in java.util`”¹².

- The programmer confirms the edit and the tool re-runs its analysis and determines that the `@Starts("nothing")` promise on the `getDescendantContainerQueues()` method is consistent with its implementation, contingent upon the consistency of the assumption.

The use of `@Assume` allows the tool user to make reasonable assumptions about libraries used by their code that have not been annotated by other means, such via standoff annotations in XML¹³. The use of `@Assume`, similar to `@Vouch`, has the potential for abuse, however, the tools ability to track each assumption and which verification results depend it make it straightforward for teams to audit the assumptions and check each for reasonableness.

Toggling analyses on and off

The ability, through the tool preferences, turn any verifying analysis on or off has been critical to the successful completion of many of the field trials of the JSure tool. For example, the uniqueness analysis is often turned off in the tool during time-constrained field trials (because, as is discussed in Chapter 4, it can be somewhat unpredictable and time consuming). This speeds analysis turnaround time for large bodies of code and improves the pace of programmer modeling—and the overall pace of the time-limited engagement. The tool user is comfortable with this trade-off because red dots decorate any verification result that is contingent upon promises that, instead of being checked by the uniqueness analysis, are trusted to be consistent with the code base.

1.5 Issues of adoption

The capture and expression of design intent is not a simple problem from the standpoint of programmers actually adopting analysis-based verification and associated tools. Immediate benefit must be visible from simple efforts by a developer or the tool is likely to become unused

¹²Our syntax, as presented in Section 3.5.4 avoids the ambiguity between package names, type names, and nested type names in patterns that is introduced if they are separated by a “.” through a reversal of the constituent portions of the declaration’s name and the use of the keyword “`in`”. Leaving out an “`in`” clause indicates to match everything, *e.g.*, if we left off the `in java.util` clause in the example above then the assumption would be made on any class named `ArrayList` in any package.

¹³Many of the classes in the Java standard library, including `java.util.ArrayList`, have been annotated by us to aid tool users. We removed the annotations on `java.util.ArrayList`, made via XML, for this example.

“shelf-ware”. Design intent must be captured incrementally and be immediately useful even when incomplete and inconsistent—guiding the source code base to (1) be consistent with design intent, (2) capture further design intent, (3) and to support, and not impede, ongoing functional evolution of the software system. Thus a key hypothesis of our approach is that the incremental model, representing an incremental return on investment (ROI), will facilitate adoption. In this section we suggest several key adoption issues and then discuss how our approach addresses them.

A key issue is to allow *late-lifecycle adoption*. It is often difficult to know *a priori* which software systems will end up becoming mission or business-critical. This imposes a fundamental design constraint on our model semantics:

In the absence of an explicit model, language-correct code is never considered “wrong”

This “rule” establishes our basis to do verification in a manner consistent with the incremental reward principle even when starting with large systems. We view an unannotated program, *i.e.*, lacking any expression of design intent, as a program that has no models against which it can be verified. Unlike similar tools for lock use policy assurance, such as RaceFreeJava [45, 46] and Guava [7], we do not require that the entire program to be assured thread-safe at once. These two tools use modular type systems that allow the program to be analyzed on a per-class basis, but require the whole program to be annotated before meaningful analysis can be performed. Tools focused more on checking, or “bug hunting”, such as ESC/Java [38, 77], by assuming unlikely invariants (*e.g.*, all fields must be non-null) force large amounts of annotation to convince the tool that correct code is not wrong in the hope of discovering defects during this process.

Our self-imposed constraint to use “language-safe-defaults” is not a prohibition against the use of inference. It simply places inferred results in a distinct category from verification results. Inference can help direct user focus to areas of the code where modeling may provide benefit or help the user complete a partially constructed model of intent. Verification results are always with respect to explicit models—models a programmer wants to remain consistent with code during system evolution. Finally, the programmer, using the @Promise scoped promise, can “change the default.”

A second key issue is *maintenance of consistency*—when inconsistent; we must provide a stepwise path to consistency. Of course, the programmer must decide where the adaptation is needed (*i.e.*, is the design intent wrong or is the code wrong?) or when inconsistency should be tolerated for a period of time. Software modeling techniques such as the various constituents of UML [49] and its precursors [19, 97], while embracing certain aspects of this approach, are not always rigorous in meaning or precisely linked to code. This is no surprise—much of the early work in modeling was motivated by “synthetic” activity (development and design), with less regard for “analytic” activity (assurance and maintenance). The distinction drawn here is between an ongoing design record (a maintained assembly of interlinked models and descriptions) and a process stepping stone on the way to a delivered implementation (intermediate steps whose relevance fades once we’ve moved to the next stage). The focus of analysis-based verification is on implementation-level design decisions, incrementality, and tool-supported consistency management.

A third key issue is to *enable verification of design intent without change to the programming language*. A trend throughout the evolution of software engineering is what we term the “rising tide of abstraction.” By this we refer to the steady increase in the level of abstraction

of programming languages over the last 40 years. Some examples include the shift from `goto` programming (ad hoc control structures) to structured programming [39], the migration from programming-in-the-small to programming-in-the-large [37], and the migration from weak to strong type systems within programming languages [93]. When the “rising” of this “tide” actually happens in mainstream practice, however, is an unpredictable market phenomenon and often sporadic. Hence, tool-supported verification without language change can have a more immediate impact and present less of an adoption barrier than achieving the same result by modifying a programming language. The focus of analysis-based verification is on the verification of program properties that are currently ignored by widely used programming languages (or handled only by runtime exceptions). Although it is possible that some of the mechanical design intent we express as annotations, (*e.g.*, Greenhouse [53], Sutherland [103]) could be expressed in a future programming language—this does not change the need for or utility of our approach to “rise” above the current level of the “tide”.

Adoption of verification (for positive assurance) can be difficult because, ironically, it is less immediately useful to developers than bug finding. Its greater value, as often perceived by developers, is in facilitating update and evolution of code in a way that can be more assuredly safe. This is because it documents key design decisions and provides immediate feedback (during the update/evolution process, later) regarding the consistency of potential changes with established design intent (“established” because it was actually verified during initial development).

1.6 Claims and contributions

In this section we present the contributions of this thesis. We also differentiate our work from the work of other members of the Fluid project at Carnegie Mellon University. We first discuss the overall vision of the Fluid project followed by a presentation of our contributions toward reaching this vision. We conclude with a summary of the evidence we present, throughout this dissertation, in support of the claims of this research.

1.6.1 The vision of the Fluid project

The key idea and overall vision of the Fluid research group is focused *analysis-based verification* for software *quality attributes*⁽¹⁾ as a *scalable*⁽²⁾ and *adoptable*⁽³⁾ approach to the *verification*⁽⁴⁾ of consistency of code with its *design intent*⁽⁵⁾. Each of the superscripts in the previous sentence is elaborated below.

- (1) **Quality attributes:** Focus on narrowly-targeted “mechanical” requirements related to quality attributes—what we term *mechanical program properties*—rather than functionality. These include safe concurrency with locks, data confinement to thread roles, static layer structure, and others.

Verification of each quality attribute listed above has its own combination of contributing constituent static analyses. For example, safe concurrency with locks is verified through a combination of underlying low-level constituent analyses: lock policy, must-hold lock, must-release lock, may-equals, uniqueness, effects upper bounds, and binding context.

- (2) **Scalable:** Adapt constituent static analyses to enable composition—allowing tools to support “separate verification” in a manner similar to separate compilation. Composition is key to the ability to scale up, in terms of code size, to real-world software systems.
- (3) **Adoptable:** Offer a “gentle slope” with regard to return on investment (ROI) for developers and teams for their effort. Any increment of effort we ask programmers to undertake should yield a generally immediate reward in the form of added assurance, expression of a model, guidance in evolution, or bug finding. We refer to this as the *incremental reward principle*.

We achieve adoptability through attention to the management of proof structures, and through careful engineering of tools and teams. We set for ourselves a goal that, when we visit a team of developers and their code for a few days, we can work with them to obtain useful and actionable results before lunch on the first day.

- (4) **Verification:** Sound static analyses produce no false negatives about an attribute and a model of programmer design intent. We build on sound analyses, which is how we get to *verification* results.
- (5) **Design intent:** Express programmer design intent as fragmentary models/specifications focused on quality attributes. Instead of asking the programmer to explicitly express functional specifications, we ask the programmer to record design intent related to particular mechanical program properties that may be difficult for them to manually check.

Our work, incorporating the prior sound analysis work of the Fluid project, led to the development of the JSure prototype analysis-based verification tool. The tool supports the verification of the promises listed in Figure 1.7 and Figure 1.8 and was used in our field validation. The field validation presented in Chapter 4 validates the overall vision of the Fluid project as well as our contribution. It has also informed the development of our work.

Other new technical and engineering results contributed by this thesis include:

- **The drop-sea proof management system:** As introduced in some detail above, drop-sea is the proof management system used by the JSure tool. Drop-sea manages the results reported by constituent program analyses and automates the proof calculus presented in Chapter 2 to create verification results based upon these findings.
- **Management of contingencies—the red dot:** Drop-sea allows several unverified contingencies to exist in a chain of evidence about a promise. A programmer can vouch for an overly conservative analysis result—changing it from an “x” to a “+”. A programmer can turn off a particular program analysis causing all the promises checked by that analysis to have no results—causing the tool to trust these promises without any analysis evidence. Finally, the programmer can assume something about a component that is outside of the programmer’s scope of interest (*e.g.*, on the other side of an organizational or contractual boundary). These actions introduce a contingency into any proof that relies upon them. Drop-sea explicitly tracks these contingencies and flags them with a red dot.
- **Proposed promises:** Our approach has constituent analyses report any necessary prerequisite assertions as part of each analysis result. Analyses, when they report a prerequisite assertion, propose promises that may or may not exist in the code. A special analysis called *promise matching* is used to “match” each proposed promise with a programmer-expressed promise in the code. If no “match” can be found, *i.e.*, a promise proposed by a constituent analysis is not in the code base, then the computation that produces verification results is able to use the unmatched proposed promises to determine the “weakest” prerequisite assertion for each promise in the code base. This allows the tool to propose “missing” annotations, from the point of view of the constituent analyses, to the code that can be reviewed and accepted by the tool user.
- **Scoped promises:** Scoped promises are promises that act on other promises or analysis results within a static scope of code. We introduce three types of scoped promises: `@Promise` to avoid repetitive user annotation of the same promise over and over again in a class or package, `@Assume` to support team modeling in large systems where programmers are not permitted access to the entire system’s code, and `@Vouch` to quiet overly conservative analysis results. Scoped promises help to “scale up” the ability of a programmer or a team of programmers to express design intent about a large software system.
- **An approach to the specification and verification of static program structure:** Previous work by Murphy and Notkin [85] has demonstrated the utility to practicing programmers of tool support to understand and maintain structural models of their code. Our approach combines the creation of the high-level model and a mapping from the high-level model to the source code via source code annotations—primarily a syntactical difference—but our purpose is the same: to help programmers express,

understand, and maintain the static structure of their code. Our primary contribution to prior work is the addition of a lightweight approach to specify and verify static layers with well defined semantics that we suggest are consistent with traditional layered semantics. In addition, our approach, as presented in Chapter 6, more naturally facilitates composition of multiple overlapping static models.

The JSure tool supports the Java programming language. Hence, our results support the above claims for the Java programming language. Proof of further external validity, *e.g.*, to other similar programming languages, while a reasonable inference, is outside the scope of this work. We do conjecture that our results can be extended to other languages with strong typing such as C# or Ada95. It is more speculative to contemplate C++ and C, though there may be *idioms* that can be addressed.

Our approach is not without drawbacks. The largest is the size and complexity of the tool required to support its effective use. The JSure prototype tool is 260 KSLOC of Java code adding to the over 1,300 KSLOC of Java code comprising the Eclipse Java IDE (which JSure augments). Gaining programmer trust in such a large and opaque tool is a challenge. The “opacity” of the tool can be contrasted to the transparency of simple testing with a framework such as JUnit [14]. We are encouraged by the success in practice of large and complex quality improvement tools such as debuggers, profilers, and (more recently) automated refactoring. Large and opaque tools have had success in practice when they provide transparent benefit to the programmer with acceptable cost and usability.

1.6.3 Validation

In this section we summarize the evidence presented throughout this dissertation in support of the claims of this research. In the three sections below we summarize the evidence presented for each principal contribution of our work (Section 1.6.2). Chapter 7 recapitulates the evidence we provide in support of our thesis claims.

Meta-theory

The meta-theory developed in this thesis is presented in Chapter 2. This chapter demonstrates that our verification proof calculus supports the construction of verification proofs from fragmentary analysis results reported by multiple underlying constituent analyses. We establish soundness of our verification proof calculus by a proof (of Theorem 2.7.4) that relates a semantics of fragmentary analysis results to broad conclusions regarding the program being analyzed. Several key lemmas (Section 2.6 and Section 2.7) are proved in support of the soundness theorem about the verification proof calculus and the precise semantics of fragmentary analysis results.

User experience design and tool engineering approach

The JSure prototype tool, incorporating the prior work of the Fluid project on sound analysis, was developed as part of our work. The tool provides evidence that our approach, with its constituent analyses, is *feasible* for the tool-supported verification of non-trivial narrowly-focused mechanical properties about programs with respect to explicit models of design intent.

The engineering of this tool (and its user experience) is introduced in Chapter 1 and presented in further depth in Chapter 3.

Our engineering work provides a technical and software framework, realized in the JSure tool, for the verification of mechanical program properties. Two case studies of adding new aggregate analyses to the JSure tool are used to qualitatively evaluate the benefits, as well as identify limitations, of this verification framework (*e.g.*, drop-sea, scoped promises, proposed promises, the red dot), with respect to *scalability* when new attributes are added. These case studies are presented in Chapter 5 and summarized below in Section 1.8. One case study, *static layers*, was conducted by the author and the technical details of this analysis are presented in Chapter 6. The second case study, thread coloring, was conducted by Sutherland and the technical details of this analysis are presented in [103].

Field validation

Nine trials using the prototype JSure tool in the field on open source, commercial, and government Java systems are presented in Chapter 4. As illustrated in Figure 1.22, our field trials provide empirical evidence in support of the contribution of our work as well as the overall vision of the Fluid project. These field trials are used to

- Quantitatively measure our prototype tool's *scalability* with respect to code size.
- Demonstrate that analysis-based verification adheres to the *incremental reward principle* and can provide immediate benefit to disinterested practitioners.
- Measure the *effectiveness* of analysis-based verification with respect to defects found and perceived value of the approach by disinterested practitioners.
- Demonstrate the feasibility of *adoption* late in the software engineering lifecycle.

An overview of our trials of the JSure tool in the field is presented in the next section.

1.7 Field trials in a nutshell

Chapter 4 presents results from nine field trials of the JSure tool on open source, commercial, and government Java systems conducted by the Fluid project between July 2004 to October 2009. In this section we present a brief overview of these field trials and the results obtained from them.

Figure 1.23 provides the date and duration of each field trial as well as the client organization visited and which client software was examined. Many of these field trials were performed under strong restrictions relating to disclosure, *e.g.*, *Company-A* and *Company-B*, while others were deemed by the participating company to be more public, *e.g.*, Sun Microsystems and Yahoo! field trials on open source software (Electric and Hadoop, respectively) that they are a patron of. Although field trials do not allow as controlled an environment as case studies (typically performed by the author at the university) the feedback they have provided has been essential to the evolution of the work we present. In addition, they provide evidence that our approach is considered of value to impartial practitioners and, therefore, feasibly adoptable in practice.

Date	Duration (days)	Organization	Software Examined	Code Size (KSLOC)
Jul 2004	3	<i>Company-A</i>	Commercial J2EE Server- <i>A</i>	350
Dec 2004	3	NASA/JPL	Distributed Object Manager	42
			MER Rover Sequence Editor	20
			File Exchange Interface	12
			Space InfeRed Telemetry Facility	18
Feb 2005	3	Sun	Electric – VLSI Design Tool	140
Oct 2005	3	<i>Company-B</i>	Commercial J2EE Server- <i>B</i>	150
Jul 2006	3	Lockheed Martin	Sensor/Tracking (CSATS)	50
			Weapons Control Engagement	30
Dec 2006	1	Lockheed Martin	Equipment Web Portal	75
Mar 2007	3	NASA/JPL	Testbed	65
			Service Provisioning (SPS)	40
			Mission Data Processing (MPCS)	100
			Next-Generation DSN Array	50
Oct 2007	3	NASA/JPL	Maestro	17
			Command GUI	139
			Accountability Services Core	48
Oct 2009	3	Yahoo!	Hadoop HDFS	107
			Hadoop MapReduce	281
			Hadoop ZooKeeper	62

Figure 1.23: Date, duration, organization, software examined, and code size of the Java software systems examined during the 9 field trials of the JSure tool.

Each field trial took place at the participant’s facilities (*e.g.*, at Sun Microsystems or at Yahoo!)—typically in the work areas of the developers with whom we were collaborating—and was 3 days in duration. The largest code base examined in a field trial was 250 KSLOC with a code size of 100 KSLOC to 150 KSLOC being more typical. In addition, it was unusual for us to be allowed access to the source code that the participant wanted to examine before we arrived on the first day.

The feedback received from our field trials is encouraging. For example, Lockheed Martin Advanced Technology Laboratories published a short article about our field trials in [25]. In terms of effectiveness they report

“[JSure] identified logic and programming errors in sensor and tracking software in which earlier extensive testing had revealed no errors.”

In terms of usability they report

“[JSure] is also easy to use. After a brief training period of less than four hours, developers were able to run [JSure] on their programs with only minimum assistance.”

In terms of perceived value and effectiveness they report

“One engineer observed during the [JSure] tests, ‘I can’t think of any of our Java code I wouldn’t want to run this tool on.’ Another engineer observed, [JSure] ‘pointed out things that we would not have looked at—would not have even noticed or caught in a code review.’ ”

Critical feedback from these field trials has helped to shape the evolution of our work and the engineering of the JSure tool. For example, significant changes to the annotation syntax have been made based upon feedback from our work in the field¹⁴. One analysis—the analysis used to verify uniqueness promises—has been found to not scale well, in terms of performance, when used on real-world Java code and is being replaced in the tool.

The core technical contributions of this thesis were in place for each of the nine field trials. All nine field trials used the Eclipse-based JSure prototype tool, including the drop-sea proof management system, the @Promise scoped promise, and contingency management via the red dot (*e.g.*, turning analyses on and off, and trusted promises). @Assume and @Vouch were later added to JSure in response to feedback from the field. The last capability added to JSure as part of this work was proposed promises, which was developed in response to feedback from the field to help support model expression (as discussed in Section 1.4.2), but was not available for use during any field trial.

As noted above, our field trials provide empirical evidence in support of our work as well as the overall work of the Fluid project. The constituent analyses developed by Greenhouse, Boyland, and Sutherland could not be employed effectively to obtain results meaningful to professional software developers without the contribution of our work. This is because they had no significant support for user interaction, and, more importantly, could not create proof structures that rendered results directly meaningful to developers. For example, the proof structures developed over the course of three days during our first field trial include several thousand nodes in proof trees and several thousand individual underlying constituent analysis results. There is no means for individual developers (or the research team, for that matter) to manage this quantity of separate proof elements without tool assistance.

A brief aside about using dynamic analysis to help understand concurrency

Our experience in the field drove the development of the Flashlight dynamic analysis tool [58]. This tool observes the concurrency attributes of a running program (*e.g.*, shared state, lock use, lock ordering, and lock contention) and provides a query-oriented user interface. In addition, Flashlight is able to propose annotations to the code that can be verified by JSure, *i.e.*, propose promises based upon what it observed in terms of the program’s locking behavior.

The Flashlight dynamic analysis tool was developed because many programmers did not understand the concurrency that was occurring within code that they were tasked to maintain. Their company might have purchased the code, the original developers might have been reassigned to other projects, or they didn’t understand of the concurrency policies that the libraries and frameworks that they used imposed upon their code.

¹⁴It is interesting to compare the annotations used by Greenhouse in [53] to those in this dissertation—the evolution of the annotation syntax since 2003 is apparent.

1.8 Case studies of constructing new aggregate analyses in a nutshell

Chapter 5 presents two case studies of adding new aggregate analyses to the JSure prototype tool: *thread coloring* and *static layers*. These case studies are used to evaluate the verification framework (*e.g.*, drop-sea, scoped promises, proposed promises, the red dot) developed as part of this work and implemented in the JSure tool with respect to the benefit the framework provides for the incorporation of new analyses.

Sutherland thread coloring

Sutherland, using our approach to analysis-based verification, has developed an approach, called *thread coloring*, that allows developers to concisely document their thread usage policies in a manner that enables the use of sound scalable analysis to assess consistency of policy and as-written code [104, 103, 105]. Thread coloring is a useful technique to statically verify concurrency policies that do not involve locking, such as the thread-confinement policy adopted by most object-oriented GUI frameworks (*e.g.*, SWING/AWT and SWT).

Sutherland’s work was done simultaneously with the work presented in this thesis and provides a case study of the use of our approach for the verification a new mechanical program property. In particular, drop-sea and the @Promise scoped promise were found to be beneficial to communicate verification results to the tool user and to reduce repetitive annotation, respectively.

Static program structure—static layers

Chapter 6 presents an approach to the specification and verification of static program structure developed by the author. Our approach builds upon prior work by Murphy and Notkin [85], called *reflexion models*, that demonstrated the utility to practicing programmers of tool support to understand and maintain structural models of their code.

This work is presented, in the context of this thesis (and as a case study in Chapter 5), as evidence that our approach is scalable with respect to analysis breadth. In addition, this work is presented as evidence that our approach supports the verification of a non-concurrency related mechanical program property.

1.9 Related work

The motivation and setting of our work draw from the sound analyses developed by the Fluid research group led by William L. Scherlis at Carnegie Mellon University. This includes sound analysis techniques for the verification of lock policies by Greenhouse [53], object-oriented effects by Greenhouse and Boyland [54], and unique references by Boyland [20]. We build upon this body of prior work, by addressing the limitations discussed above, to take a first step towards increasing its practicability and adoptability. Further, our work has helped to facilitate trials of analysis-based verification in the field.

As discussed above, heuristics-based static analysis tools, such as FindBugs [62], have

recently started to support the use of annotations to control false positive results and enable improved bug finding. Despite the demonstrated usefulness of these tools, our approach is focused on enabling tool-assisted verification rather than finding code defects (although finding code defects does occur as a programmer works to attain model-code consistency). We suggest that these tools, in the future, could benefit from our work. In particular, they could use our approach to enable the sound verification of annotations with precise semantics.

A contemporary project focused on practical specification and verification is the **Spec#** project at Microsoft Research¹⁵. This project builds upon the results of the ESC/Modula 3 and ESC/Java projects [38, 47]. This research has produced a C#-like language that allows the first-class expression of preconditions, postconditions, and invariants [10]. The primary technical contribution of this project is the **Boogie** verifier for object-oriented programs [11]. Similar to the **JSure** analysis-based verification system, the **Spec#** compiler and **Boogie** verifier are built into a widely-used IDE, in the case of **Spec#** Microsoft Visual Studio (rather than Eclipse). While we suggest that the vision of this work is the same as ours, namely to find widespread adoption in mainstream practice, their approach differs from ours in several ways. First, their approach requires programmers to express preconditions, postconditions, and invariants about their code. We have purposely avoided this requirement because of the large number of programmer annotations required in even modest-sized code bases. We agree, however, that if they are successful, the widespread specification of preconditions, postconditions, and invariants would have a positive impact on mainstream software quality practice¹⁶. Second, their approach centers around the **Boogie** verifier. The **Spec#** compiler translates the annotated program into **BoogiePL**, an intermediate language for program analysis and program verification [35], that is then processed by the **Boogie** verifier. This differs from our approach where we provide a core proof management system, drop-sea as described in Chapter 3, that allows “plug-in” constituent verifying analysis. In our approach the constituent analyses must encode a deep understanding of the programming language semantics, in theirs, this must be expressed in **BoogiePL** for processing by the **Boogie** verifier. Finally, their focus on assertions about the program’s state makes it difficult to express more “mechanical” attributes of a program’s design, *e.g.*, non-state related assertions, such as the specification and verification of static program structure (introduced in Chapter 6) or the Fluid module system [103].

The Java Modeling Language (JML)¹⁷ [74, 75] provides a specification language for the Java programming language, but unlike our approach, its focus is on behavioral or functional specification. JML combines the design by contract approach of Eiffel [81] and the model-based specification approach of the Larch family of interface specification languages [57], with some elements of the refinement calculus [41]. The JML is supported by a large research community as well as by several tools [76]. One example verification tool, the LOOP tool [67] uses JML for a specification language and supports verification through PVS [91]. LOOP, like many of these tools (*e.g.*, KeY [15] and JIVE [84]), requires manual interaction with the theorem prover to attain verification. Our approach supports significantly more automation but is limited to the verification of narrowly-targeted quality attributes. By adopting this narrow focus we are able to avoid requiring that the programmer use only a subset of the Java programming language.

¹⁵<http://research.microsoft.com/en-us/projects/specsharp/>

¹⁶Stranger things have happened, Kent Beck made unit testing popular by referring to it as “Extreme” [13, 14].

¹⁷<http://www.eecs.ucf.edu/~leavens/JML/>

Several programming languages, similar to `Spec#`, support specification as part of the language. One example that has been successfully commercialized by Altran Praxis¹⁸ is SPARC Ada [9]. The use of SPARC Ada, or similar languages designed with verification in mind, require an up-front commitment in terms of cost and schedule with the strategic benefit of increased code quality. Our approach differs in that we support adoption late in the software lifecycle but is limited to the verification of narrowly-targeted quality attributes.

Other related work is discussed in context throughout the presentation of analysis-based verification in this thesis.

1.10 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 develops a formal model of our approach to analysis-based verification. This model is the basis of the tool engineering we present in Chapter 3; however, several versions of the JSure prototype tool were developed prior to this model and influenced its development. We choose to present this model first to precisely describe the essential elements of our approach and to assure the reader that the tool engineering is based upon principled foundations.

Chapter 3 presents significant details about the design and engineering of the JSure prototype tool. The realization of this tool has evolved based upon feedback from its use in several field trials (discussed in Chapter 4). The chapter presents an architecture for an analysis-based verification tool, the drop-sea proof management system, the management of contingencies—the red dot, the scoped promises `@Promise`, `@Assume`, and `@Vouch`, and the JSure tool user interface.

Chapter 4 presents details about the nine field trials conducted by the Fluid project over a number of years with the evolving JSure prototype tool. These field trials are presented as evidence of the practicability and adoptability of analysis-based verification.

Chapter 5 presents two case studies (thread coloring and static layers) of adding new analyses to the JSure tool. These case studies evaluate the verification framework (*e.g.*, drop-sea, scoped promises, proposed promises, the red dot) developed as part of this work and implemented in the JSure tool to scale up with respect to analysis breadth.

Chapter 6 presents an approach for the specification and verification of static program structure. Our approach builds upon previous work by Murphy and Notkin [85] that demonstrated the utility to practicing programmers of tool support to understand and maintain structural models of their code. This work is presented, in the context of this thesis, as evidence that analysis-based verification is scalable with respect to new assurance attributes, in particular non-concurrency related program attributes.

Chapter 7 recapitulates the evidence we provide in support of our thesis claims. In addition to the field trials and case studies of incorporating new analyses presented in Chapter 4 and Chapter 5, respectively, a cost-benefit analysis is presented in support of the thesis claims.

Chapter 8 concludes with a summary of the contributions of this research and a discussion of future work.

¹⁸<http://www.sparkada.com/>

Foundations

“On two occasions I have been asked [by members of Parliament!], ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”

— Charles Babbage

2.1 Introduction

In this chapter we develop a formal model of our approach to analysis-based verification. This model is the basis of the tool engineering we present in the next chapter; however, several versions of the JSure prototype tool were developed prior to this model and influenced its development. We choose to present this model first to precisely describe the essential elements of our approach and to assure the reader that the tool engineering is based upon principled foundations.

The contribution of this chapter is a formal model that describes the construction of proofs within our approach. This includes specifying precisely how sound “plug-in” program analyses report their findings and the development of an automatable proof calculus to create program- or component-level results based upon these findings. Our approach supports separate analysis of components and allows composition of the results such that the outcome corresponds to that of a whole-program analysis. The requirements to support modularity and composability permeate the formal systems presented in this chapter. Our main technical result is a soundness theorem that relates the proof calculus to a semantics for analysis results.

In a sense, sound combined analyses for analysis-based verification is intended to do for program analysis what Nelson and Oppen did for theorem proving through the introduction of cooperating decision procedures [86]. Our approach allows subsidiary program analyses to cooperate in a similar manner. Rather than combining decision procedures, we combine program analyses into a single system with well-defined semantics. Indeed, in our approach, knowledge of the semantics of programming languages is embodied entirely in these subsidiary program analyses, and the logic serves to soundly combine fragmentary analysis results at a level of abstraction and aggregation appropriate for programmers and other human users. Both approaches levy requirements on constituent capabilities (*i.e.*, decision procedures for Nelson and Oppen and verifying analyses for our approach) and so are “semantically scalable”

```

1 @Stateless
2 public class TravelAgentBean implements TravelAgentRemote {
3
4     @PersistenceContext(unitName="titan") private EntityManager manager;
5
6     @Starts("nothing")
7     public void createCabin(Cabin cabin) {
8         if (findCabin(cabin.getId()) == null)
9             manager.persist(cabin);
10    }
11
12    @Starts("nothing")
13    public Cabin findCabin(int pKey) {
14        return manager.find(Cabin.class, pKey);
15    }
16 }

```

Figure 2.1: A EJB 3.0 stateless session bean that promises, via the `@Starts("nothing")` annotations at line 6 and 12, not to start any threads. This promise is consistent with the programming restrictions placed on stateless session beans by the EJB 3.0 Specification.

in this respect.

To introduce our approach and sketch its major components we start with a motivating example drawn from a Java EE system. This example is followed by several sections that develop our formal model using a running example from the `util.concurrent` library. Once we have fully presented our approach to proofs within an analysis-based verification system we continue by introducing a semantics of analysis results and prove a soundness theorem. We end the chapter with a short discussion about the differences between our proof calculus and a Hoare logic. In particular, we discuss the reasons why our calculus would not, traditionally, be considered a Hoare logic despite notational similarities.

2.1.1 A motivating example

We start with a program that has been annotated with extra-linguistic programmer design intent, that we refer to as *promises*. An example of a program containing promises is shown in Figure 2.1. `TravelAgentBean` is an EJB 3.0 stateless session bean from the Titan Cruises Java EE application [24].

The EJB 3.0 Specification [36] states that stateless session beans are not allowed to start any threads, however, this constraint is not statically enforced by Java. Using our annotations, each method promises that it never starts a thread. The `@Starts("nothing")` annotation at line 6 promises that the `createCabin` method will not start a thread. The `@Starts("nothing")` annotation at line 12 promises that the `findCabin` method will not start a thread. The `@Stateless` and `@PersistenceContext` annotations are provided for use by the EJB 3.0 container and are used to identify the enterprise bean type and to inject a connection to a database, respectively. The `@Stateless` and `@PersistenceContext` annotations are not promises checked by analysis-based verification.

The programmer, after entering a few promises, wants to know, “Is my code consistent with these promises?” and “If not, what should I do next?” Analysis-based verification answers

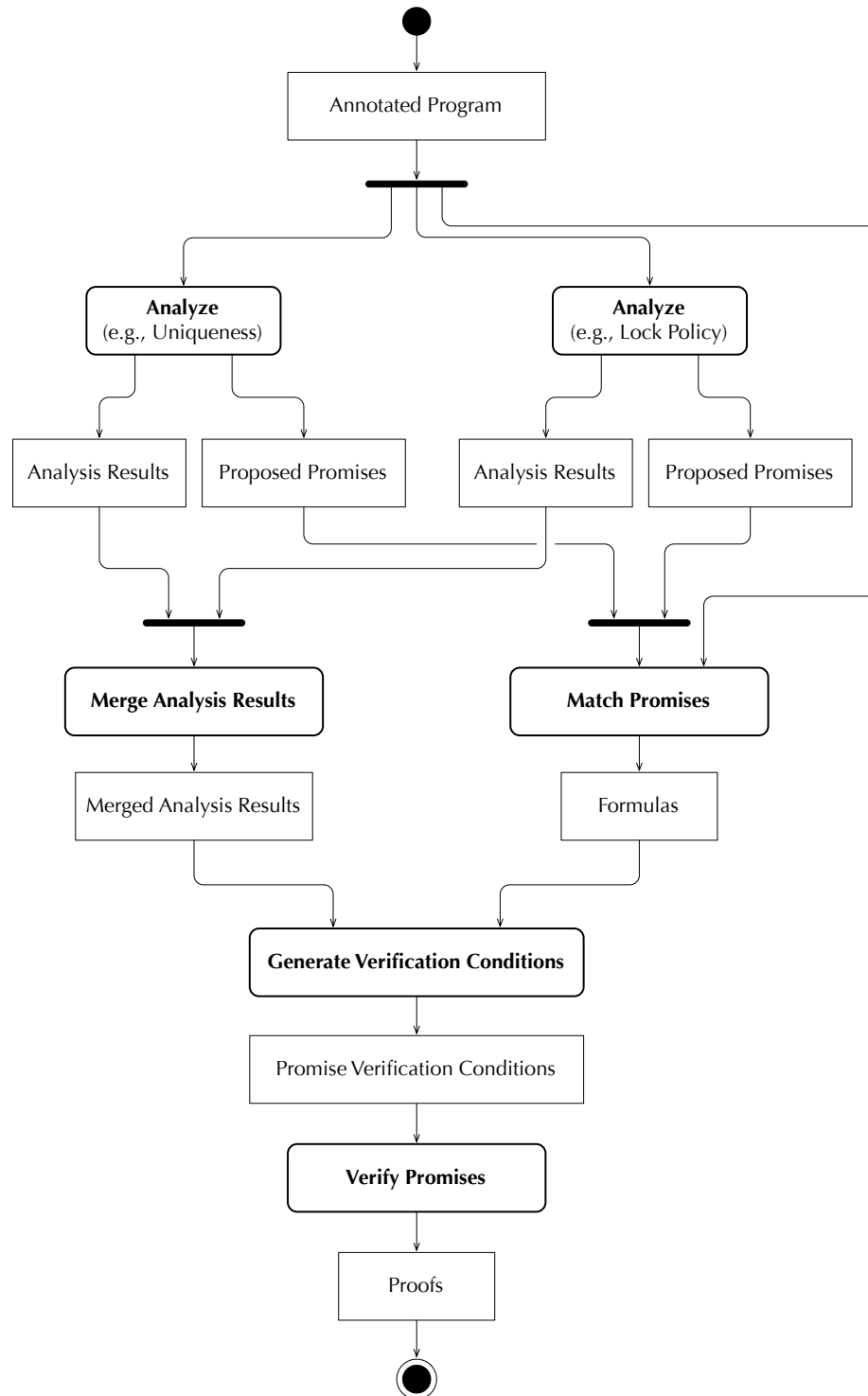


Figure 2.2: An activity diagram describing the flow of events during analysis-based verification.

Analysis Results for TravelAgentBean

	Finding	About	Prerequisite	Description
f_1	+	r_6	\top	<code>createCabin</code> starts no threads
f_2	+	r_6	q_1	<code>findCabin</code> promises it starts no threads
f_3	+	r_6	q_2	<code>getId</code> promises it starts no threads
f_4	+	r_6	q_3	<code>persist</code> promises it starts no threads
f_5	+	r_{12}	\top	<code>findCabin</code> starts no threads
f_6	+	r_{12}	q_4	<code>find</code> promises it starts no threads

	Proposed Promises	
	Promise	On
q_1	<code>@Starts("nothing")</code>	<code>findCabin(int)</code> at line 13
q_2	<code>@Starts("nothing")</code>	<code>getId()</code> in class <code>Cabin</code>
q_3	<code>@Starts("nothing")</code>	<code>persist(Object)</code> in class <code>EntityManager</code>
q_4	<code>@Starts("nothing")</code>	<code>find(Class, int)</code> in class <code>EntityManager</code>

Figure 2.3: Analysis results (top) and proposed promises (bottom) for the `TravelAgentBean` compilation unit in Figure 2.1.

these questions by performing the activities in Figure 2.2 that result in a report to the programmer about which promises are consistent with the program’s code and which are not. This report summarizes (*i.e.*, is a query of) the verification proofs produced by the activities in Figure 2.2. If a promise is inconsistent because the programmer needs to provide further design intent (*i.e.*, enter more promises), then a set of *proposed promises* is provided to the programmer for consideration.

Each of the activities in Figure 2.2 is sketched below and detailed in the subsequent sections of this chapter.

1. **Analyze:** A set of analyses is run over the program. Figure 2.2 shows two, uniqueness and lock policy, but typically more are included (as listed in Figure 1.3 on page 8). Analyses, in our approach are “plug-in,” in the sense that more can be added or existing ones can be replaced. Each analysis produces two outputs: a set of results and a set of proposed promises. It is the job of an analysis to check promise–code consistency for each promise it is responsible for. If we refer to the `@Starts("nothing")` promise at line 6 as r_6 and the `@Starts("nothing")` promise at line 12 as r_{12} then the analysis responsible for checking these promises, called thread effects, would produce the tables in Figure 2.3. Each analysis result, f_1 through f_6 , reports a consistency finding with respect to the code, “+” for consistent or “x” for inconsistent; a promise that the consistency finding is about; a prerequisite assertion that the result is contingent upon; and an explanation of the result for the programmer. Multiple results about the same promise, *e.g.*, f_1 through f_4 about r_6 , are later conjoined. The conjoined prerequisite assertion for r_6 is $\top \wedge q_1 \wedge q_2 \wedge q_3$. The symbol \top , which represents the tautology, is used to express the lack of a prerequisite for an analysis result, *e.g.*, f_1 .

Each result indicates a “point of consistency” (+) or “point of inconsistency” (x) between the promise and the code. The reported prerequisites may make that result contingent

upon the consistency of other promises—using the other promises as a *cut* or cut-point to avoid whole-program analysis. However, an analysis typically does not check if a promise that it reports as a prerequisite exists in the code, rather it *proposes* that such a promise should exist¹. For presentation purposes we use r to refer to promises that appear in the code, q to refer to proposed promises, and f to refer to analysis results.

Each proposed promise, q_1 through q_4 in Figure 2.3, specifies an assertion and a location in the code for that assertion. A particular proposed promise may or may not be able to be matched with a promise in the code, *e.g.*, the proposed promise q_1 can be matched with the promise r_{12} in the code. Each proposed promise is created to be used as the prerequisite for an analysis result, *e.g.*, q_1 through q_4 are, and must be, referenced by the prerequisite of an analysis result.

Proposed promises allow a principled way to represent what is required, in terms of design intent, for a promise to be made consistent. The programmer, aided by proposed promises, can interact with the tool toward a complete model of design intent after providing just a few annotations.

Our approach to collecting analysis results and associated proposed promises allows analyses to be run in parallel. This is illustrated in Figure 2.2: the uniqueness and lock policy analyses are accomplished concurrently. In addition, it allows examination of the compilation units that make up a program by a particular analysis to be run in parallel. Parallelism is allowed because no analysis modifies the annotated program it is examining and each reports into their own tables that are later merged.

2. **Merge Analysis Results:** All the reported analysis results are collected into a single table that represents the complete set of analysis results about the set of compilation units examined. In our example, we have only analyzed one compilation unit, `TravelAgentBean`, with one analysis, thread effects, and the table of analysis results in Figure 2.3 is also our merged results table. We call the set of merged analysis results R .
3. **Match Promises:** A special analysis attempts to match each promise that was proposed by an analysis to a *real promise* in the program. Real promises are promises that appear in the program, *i.e.*, they are programmer-written. The output of the promise matching process is a set of formulas, which we call Φ , that reflect that a real promise implies a proposed promise. This means that if the real promise is satisfied then the proposed promise is also satisfied. For `TravelAgentRemote`, we find that r_{12} implies q_1 , and produces the formula $r_{12} \rightarrow q_1$. We don't know if real promises exist for q_2 , q_3 , or q_4 because, in this example, we haven't examined the code of the `Cabin` or `EntityManager` classes. Therefore, the output of promise matching is

$$\Phi = \{r_{12} \rightarrow q_1\}.$$

A straightforward intuitionistic propositional logic, which we call *promise logic* (\vdash_{pl}), is used to “substitute” real promises for proposed promises during promise verification. The details about promise matching are presented in Section 2.4.

4. **Generate Verification Conditions:** Based upon the set of merged results and the set of formulas produced by promise matching, we can generate a set of promise verification conditions. We call this set V . Each promise verification condition is a triple

¹Our approach *does* allow an analysis to report a real promise as a prerequisite. Therefore, the use of proposed promises is not essential to the soundness of our approach. This is discussed further in Section 2.3.1.

that states the consistency of a real promise (right) is contingent upon the consistency of a formula of promises (left). The middle of the triple is a tuple containing the set of analysis results that were reported about the real promise (right) and the (whole) set of promise matching formulas, Φ . For `TravelAgentRemote`, two verification conditions are generated. The first is the verification condition for the consistency of the `@Starts("nothing")` at line 6

$$\{\top \wedge q_1 \wedge q_2 \wedge q_3\} (\{f_1, f_2, f_3, f_4\}, \Phi) \{r_6\}$$

and the second is the verification condition for the of the `@Starts("nothing")` at line 12

$$\{\top \wedge q_4\} (\{f_5, f_6\}, \Phi) \{r_{12}\}.$$

The details about verification condition generation are presented in Section 2.5.1.

5. **Verify Promises:** Using the set of promise verification conditions a proof calculus (\vdash_{coe} where *coe* stands for a “chain of evidence”) is used to reason toward a goal prerequisite assertion of \top meaning that the real promise is consistent with the code. For example, the sequent

$$V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}$$

means that, based upon the set of verification conditions, V , generated from the analysis results about the `TravelAgentBean` compilation unit, the `@Starts("nothing")` promise, r_6 , at line 6 in Figure 2.1 is consistent if the proposed promises q_2 , q_3 , and q_4 are consistent². This contingent result allows the programmer to proceed by either turning the proposed promises into real promises or by trusting the consistency of the proposed promises. The details about this calculus are presented in Section 2.5.

2.1.2 Interpreting verification results

A particular promise about a program is either *consistent* or *inconsistent* with respect to that program’s implementation. Analysis-based verification is used to construct (and explain) a proof of promise consistency. If our approach is unable to build such a proof for a particular promise, p , we cannot claim that p is inconsistent with respect to the program’s implementation. The reason for this, in essence, comes back to Rice’s theorem [95] which states that any non-trivial question you ask about code can be reduced to the halting problem. To avoid the lurking specter of non-computability, the program analyses that our approach uses are *conservative*—they will never report that a promise is consistent when, in reality, it is inconsistent. But they can fail to report that a promise is consistent when, in reality, it is.

Through the use of conservative program analyses and the construction of consistency proofs (as described below) analysis-based verification produces *sound* verification results. This means that if our approach reports that a promise is consistent then the assertion made by that promise is an invariant of the program, *i.e.*, it holds for all possible executions of the program.

In the parlance of program analysis tools, this means that for the analysis findings that are reports of potential inconsistency between stated promises and a program, there are no false negatives. That is, if there is an actual inconsistency, then there will necessarily be a corresponding finding. On the other hand, there may be false positives, in the sense that there may be failures to find proofs of actual consistencies.

²A proof of the sequent $V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}$ is shown in Figure 2.21.

2.1.3 A running example

The remainder of this chapter details the formal foundations of our approach to analysis-based verification using the code in Figure 2.4 from the `util.concurrent` library as a running example. This library, authored by Doug Lea and described in [73], provides components that are useful when building concurrent applications in Java, and has been included as part of the Java standard library since Java 5. To illustrate specific points of our approach, the code in Figure 2.4 contains a large number of promises—more than are required to verify its locking policy. The semantics of the promises used in the code in Figure 2.4 are explained in Appendix A, however, to make the presentation self-contained, we sketch their meaning below.

The region `Variable`, declared at line 3 with the `@Region` annotation, declares a region of the program’s state that includes, via the `@InRegion` annotation at line 16, the field `value_`. Accesses of this state are asserted to be protected by holding a lock on the field `lock_` by the `@RegionLock` promise at line 4. The `@RegionLock` promise names this locking model `VarLock`.

We have elided the code in our `util.concurrent` example to focus on the issue of object construction in the presence of the `VarLock` model. In concurrent Java code, objects are typically thread-confined when they are constructed then safely published to other threads. This means that, if we can verify that the constructor is thread-confined, we can read and write the protected state (*i.e.*, the field `value_`) without holding the lock. There are two ways that this can be established. The first is to verify that the receiver under construction is not aliased during object construction (see Section A.1.4). This is asserted by the `@Unique("return")` promise. In particular, when we can verify that the constructor does not create such an alias, it also knows that it is impossible for another thread to obtain a reference to the object under construction during the constructor’s execution. Because our analysis is modular, each constructor in the construction chain, `Object` to `SynchronizedVariable` to `SynchronizedBoolean`, must make this assertion. Notice that `@Unique("return")` does not hold for the `SynchronizedVariable` constructor because its code aliases the receiver at line 12.

The second way that we can verify that the constructor is thread-confined is by checking that it doesn’t start any threads and that it doesn’t write to any state outside of its own fields (see Section A.2.1). The `@Starts` and `@RegionEffects` promises on the constructors in the code make this claim. Any such constructor cannot pass a reference to the new object to a preexisting thread because it does not write to any objects that existed prior to the invocation of the constructor. It can write a reference to the new object to other objects created during execution of the constructor, but because it cannot start any threads, such a reference cannot be read by another thread. Using this approach, the claim that the constructors keep the state under construction thread-confined does hold.

The promises described above and the program analyses used to verify them are predominantly prior work. Analyses designed to support this composable annotation-based approach have been developed by members of the Fluid project at Carnegie Mellon University over the past decade. The region concept and effects promise, `@RegionEffects`, were developed by Greenhouse and Boyland [54]. The region promises and lock policy promises, *e.g.*, `@Region` and `@RegionLock`, were developed by Greenhouse in support of lock analysis [53]. The alias promise, `@Unique`, and associated uniqueness analysis were developed by Boyland [20, 21]. The thread effects promise, `@Starts`, and its associated analysis were developed by the au-

```

1 package EDU.oswego.cs.dl.util.concurrent;
2
3 @Region("protected Variable")
4 @RegionLock("VarLock is lock_ protects Variable")
5 public class SynchronizedVariable extends Object ... {
6     protected final Object lock_;
7
8     @RegionEffects("none")
9     @Starts("nothing")
10    @Unique("return")
11    public SynchronizedVariable() {
12        lock_ = this;
13    }
14 }

```

```

15 public class SynchronizedBoolean extends SynchronizedVariable ... {
16     @InRegion("Variable") protected boolean value_;
17
18     @RegionEffects("none")
19     @Starts("nothing")
20     @Unique("return")
21     public SynchronizedBoolean(boolean initialValue) {
22         super();
23         value_ = initialValue;
24     }
25 }

```

```

26 <package name="java.lang">
27   <class name="Object">
28     <constructor>
29       <RegionEffects>none</RegionEffects>
30       <Starts>nothing</Starts>
31       <Unique>return</Unique>
32     </constructor>
33   </class>
34 </package>

```

Figure 2.4: Elided Java code from the `SynchronizedVariable` and `SynchronizedBoolean` classes after adding the lock policy, object-oriented effects, thread effects, and uniqueness promises required to assure the locking policy of these classes. Also shown are promises about the no-argument constructor of the `java.lang.Object` class (the superclass of `SynchronizedVariable`). These promises are made as “standoff annotations” using XML structures because `Object` is part of the Java standard library and is typically used in binary form. Annotation via XML is equivalent to direct annotation of code.

thor.

Building upon this prior work, the approach presented in this chapter focuses on allowing diverse assertions to work together to produce a useful aggregate verification result. It focuses on making these constituent analyses “pluggable” and allowing the assertions to express semantics independent of the overall verifying analysis that aggregates them, as described in this chapter. As we stated at the start of the chapter, this includes specifying precisely how the “plug-in” program analyses report their findings and how the automatable proof calculus creates program- or component-level results based upon these findings.

A sketch of the steps performed to verify the example `util.concurrent` code is shown in Figure 2.5. The tables, formulas, and proofs shown are described in the next three sections of this chapter.

2.2 Promise logic

Promises are supra-linguistic formal annotations to programs introduced by Chan, Boyland, and Scherlis in [26]. Each promise has a *precise meaning* and constrains the implementation and evolution of the code it targets. Promises are also (typically) *modular*, meaning that the implementation constraint on the code of a promise is limited in scope.

A particular promise about a program is either *consistent* or *inconsistent* with respect to that program’s implementation. We use an intuitionistic propositional logic, which we call *promise logic*, to allow us to symbolically reason about the consistency of promises. A promise symbol, *e.g.*, p , in promise logic represents the proposition that a promise is consistent. It is not possible for us to meaningfully assert $\neg p$, because in our approach an inconsistent finding is conservative as discussed above in Section 2.1.2 (in fact, $\neg p$ is not even a well-formed formula in promise logic).

2.2.1 Syntax

Well-formed formulas in promise logic are defined using this grammar:

$$\phi \longrightarrow p \mid \top \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$

where p stands for a promise symbol. To reduce the need for parentheses, we adopt the convention that \wedge and \vee bind more tightly than \rightarrow . In addition, \wedge and \vee are left-associative, while \rightarrow is right-associative. We use lowercase Greek letters to represent promise logic formulas and uppercase Greek letters to represent sets of promise logic formulas. We refer to the set of all well-formed promise logic formulas as **PLFormula**.

For the purpose of allowing program analyses to report prerequisite assertions (described below) we define a subset of all well-formed promise logic formulas, **AOFormula** \subset **PLFormula**, which prohibits implications. Well-formed formulas in **AOFormula** are defined using the grammar:

$$\phi \longrightarrow p \mid \top \mid (\phi \wedge \phi) \mid (\phi \vee \phi)$$

To avoid parentheses, we adopt the conventions described above.

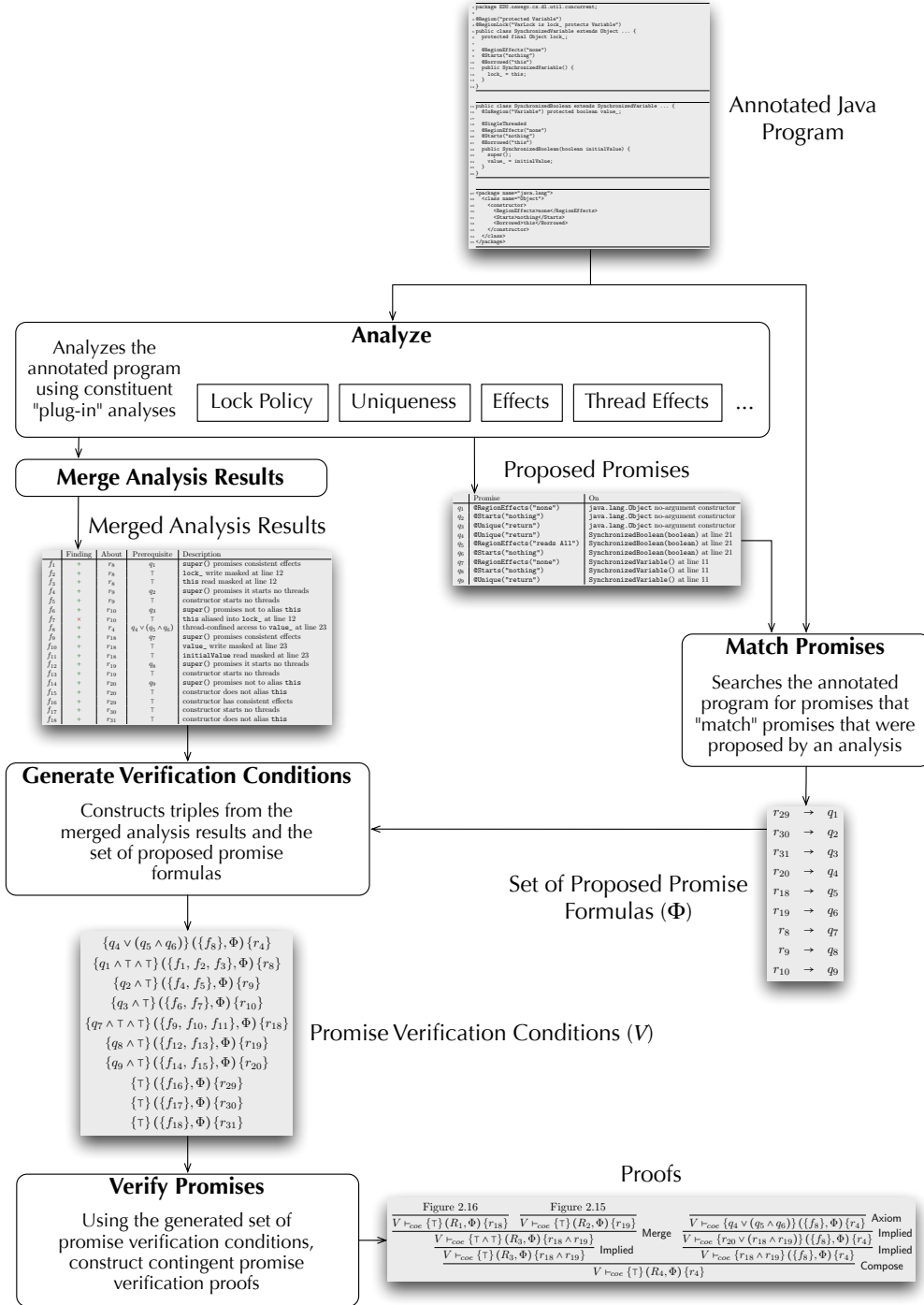


Figure 2.5: An diagram sketching the steps performed by analysis-based verification to verify the code in Figure 2.4 from the `util.concurrent` library. The arrows indicate data flow, the rounded boxes are processes. The code, tables, formulas, and proofs are the data generated and used (as described in the next three sections of this chapter). The images of the tables, formulas, and proofs give an impression of the notation and are not intended to be legible.

2.2.2 Promise symbols

We refer to each promise by a symbol. A promise may be *real* or *proposed*. A real promise appears in the code, typically as an annotation. A proposed promise is a specification for a promise (*i.e.*, an assertion and a code location) that may or may not appear in the code. We refer to the set of all promise symbols as **Promise**.

Real promises

Real promises are promises explicitly expressed by the programmer. To represent real promises in promise logic formulas, we subscript the symbol r with the line number where the promise appears within the code. Therefore, for the `util.concurrent` code in Figure 2.4, we use r_{18} to refer to the `@RegionEffects("none")` promise at line 18, r_{31} to refer to the `@Unique("return")` promise expressed in XML associated with the no-argument constructor for `java.lang.Object` at line 31, and so on.

This convention is for presentation only, promises are actually internally represented in the tool as assertions attached to nodes within a compilation unit's abstract syntax tree (AST). For example, r_{18} would be attached to the `SynchronizedBoolean(boolean)` constructor declaration within the AST for the `SynchronizedBoolean.java` compilation unit.

We refer to the set of all real promise symbols as **RealPromise**.

Proposed promises

A proposed promise is a specification, including an assertion and code location, for a promise. A proposed promise may or may not exist as a real promise. To represent proposed promises in promise logic formulas, we use the symbol q with a unique (and arbitrary) subscript. Proposed promises denote a requirement for a particular promise at a location within the program. For example, if the promises `@RegionEffects("none")` and `@Starts("nothing")` are required for the no-argument constructor for `SynchronizedVariable` then we can define the proposed promises q_1 and q_2 using the following table:

Symbol	Promise	On
q_1	<code>@RegionEffects("none")</code>	<code>SynchronizedVariable()</code> at line 11
q_2	<code>@Starts("nothing")</code>	<code>SynchronizedVariable()</code> at line 11

Again, this convention is for presentation only, proposed promises are represented as assertions that reference the AST location that they are about. For example, q_1 would be represented as an assertion with a reference to the `SynchronizedVariable()` constructor declaration within the AST for the `SynchronizedVariable.java` compilation unit.

We refer to the set of all proposed promise symbols as **ProposedPromise**.

2.2.3 Proof theory

An intuitionistic (or constructivist) sequent calculus is used for proving sequents in promise logic. We make this choice because, in addition to simplifying the logic and reducing combinatorics, it enables us to exploit a much broader range of underlying constituent analyses, in that we do not require sound approximations to both assertions and their negations. It

	Introduction	Elimination
\top	$\frac{}{A \vdash_{pl} \top} \top i$	(no elimination rule for \top – the tautology)
ϕ	$\frac{}{A, \phi \vdash_{pl} \phi} i$	(no elimination rule for ϕ)
\wedge	$\frac{A \vdash_{pl} \phi \quad B \vdash_{pl} \psi}{A, B \vdash_{pl} \phi \wedge \psi} \wedge i$	$\frac{A \vdash_{pl} \phi \wedge \psi}{A \vdash_{pl} \phi} \wedge e_1 \quad \frac{A \vdash_{pl} \phi \wedge \psi}{A \vdash_{pl} \psi} \wedge e_2$
\vee	$\frac{A \vdash_{pl} \phi}{A \vdash_{pl} \phi \vee \psi} \vee i_1 \quad \frac{A \vdash_{pl} \psi}{A \vdash_{pl} \phi \vee \psi} \vee i_2$	$\frac{A \vdash_{pl} \phi \vee \psi \quad A, \phi \vdash_{pl} \chi \quad A, \psi \vdash_{pl} \chi}{A \vdash_{pl} \chi} \vee e$
\rightarrow	$\frac{A, \phi \vdash_{pl} \psi}{A \vdash_{pl} \phi \rightarrow \psi} \rightarrow i$	$\frac{A \vdash_{pl} \phi \quad A \vdash_{pl} \phi \rightarrow \psi}{A \vdash_{pl} \psi} \rightarrow e$

Figure 2.6: The proof rules for promise logic where ϕ , ψ , and χ are promise logic formulas and A and B are sequences of promise logic formulas.

is important to note, however, that when both such sound approximations exist, the two corresponding analyses can both be exploited by the system—though their underlying logical relationship is not directly exploitable in the automated reasoning. For ease of reference, these well-known proof rules are presented in Figure 2.6. A sequent in promise logic is denoted by

$$\phi_1, \phi_2, \dots, \phi_n \vdash_{pl} \psi$$

and indicates that a proof exists from the sequence of premises $(\phi_1, \phi_2, \dots, \phi_n)$ to the conclusion (ψ) using the rules in Figure 2.6. The sequence of premises may be empty (*i.e.*, $\vdash_{pl} \psi$). We use the pl subscript on \vdash to highlight that the sequent is about promise consistency and to differentiate sequents in promise logic from sequents in our verification calculus (\vdash_{coe}) which is introduced later in this chapter. Exchange, where two members of the sequence of premises may be swapped, is implicit in promise logic.

2.2.4 Models

Not all of the annotations made to the code in Figure 2.4 are promises. Some of the annotations, rather than forming an assertion about the program’s implementation, simply give a name to some portion of the program’s code or data. For example, the annotation at line 3

```
@Region("protected Variable")
```

and the annotation at line 16

```
@InRegion("Variable") protected boolean value_;
```


define a region of the program’s state that can be referred to by the name `Variable` containing the field `value_`. Models such as this must be *well-formed*. Therefore, if the annotation at line 16 read

```
// BAD - misspelled region name
@InRegion("Variabble") protected boolean value_;
```

then it would not be considered well-formed because there is no named region called `Variable` for it to add the field `value_` into.

Both models and promises have to be well-formed and this property is readily checkable. It is checked in the JSure prototype tool by a process called “scrubbing” that is discussed in the next chapter. In this chapter we only consider well-formed models and promises. Well-formed promises, unlike well-formed models, have analysis results reported about their consistency with the code.

2.3 Analysis results

The verification of promises is performed by semantic program analyses. These analyses are modular in the following sense: Given a set of compilation units, C , containing a set of real promises, P_{real} , each analysis produces a set of results for each element of C . Each result reports a “point of consistency” or “point of inconsistency” between a promise and the code.

Definition 2.3.1 (Analysis result). An *analysis result* is a tuple

$$(\mathbf{Finding} \times \mathbf{RealPromise} \times \mathbf{AOFormula})$$

where $\mathbf{Finding} = \{+, \times\}$. We define \mathbf{Result} to be the set of all analysis result tuples.

The elements of the analysis result tuple correspond to the consistency finding, “+” for consistent or “ \times ” for inconsistent; the real promise associated with the result; and the prerequisite for the result as a promise logic formula containing no implications (*i.e.*, \rightarrow). If analyses were allowed to report prerequisites that contain implications then our approach would be unsound—because implication leads to negation appearing in prerequisite formulas. This is because in classical propositional logic, $q_1 \rightarrow q_2 \equiv \neg q_1 \vee q_2$. (This issue is further discussed in the last paragraph of Section 2.6.) Prerequisite formulas typically contain only proposed promise symbols, to allow better user reporting, but are allowed to contain real promise symbols. The prerequisite for any inconsistent result is \top because no prerequisite can lead to satisfaction.

The finding is a “sound approximation,” where “ \times ” approximates “+”, in the sense that a report of “ \times ” could represent either an actual inconsistency or a failure to prove consistency, regardless of prerequisites.

The results and associated proposed promises reported by analysis of the **Synchronized-Variable** compilation unit are shown in Figure 2.7. The figure reports analysis results tuples, for presentation purposes, in a far more readable tabular form that includes an informal description of the result, *e.g.*, f_1 represents the result tuple $(+, r_8, q_1)$. We highlight the following points about the analysis results in Figure 2.7:

Analysis Results for SynchronizedVariable

	Finding	About	Prerequisite	Description
f_1	+	r_8	q_1	<code>super()</code> promises consistent effects
f_2	+	r_8	\top	<code>lock_</code> write masked at line 12
f_3	+	r_8	\top	<code>this</code> read masked at line 12
f_4	+	r_9	q_2	<code>super()</code> promises it starts no threads
f_5	+	r_9	\top	constructor starts no threads
f_6	+	r_{10}	q_3	<code>super()</code> promises not to alias <code>this</code>
f_7	×	r_{10}	\top	<code>this</code> aliased into <code>lock_</code> at line 12

Proposed Promises		
	Promise	On
q_1	<code>@RegionEffects("none")</code>	<code>java.lang.Object</code> no-argument constructor
q_2	<code>@Starts("nothing")</code>	<code>java.lang.Object</code> no-argument constructor
q_3	<code>@Unique("return")</code>	<code>java.lang.Object</code> no-argument constructor

Figure 2.7: Analysis output for the `SynchronizedVariable` compilation unit in Figure 2.4.

- Each individual analysis result reports a “point of consistency” or “point of inconsistency” about exactly one promise.
- An analysis results is only reported about a real promise—never a proposed promise.
- Many individual analysis results may be reported about a single real promise. For example, three results are reported about r_8 . This approach works naturally with many types of program analysis, *i.e.*, they can report a result about each AST node they find consistent or inconsistent with a promise, without prohibiting a single “grand” consistency result. (Our approach, as described in Section 2.5.1, conjoins all the results about a promise into a verification condition.)
- Analysis results about the same real promise may report different consistency findings. For example, f_6 finds r_{10} to be consistent, but f_7 finds r_{10} to be inconsistent. This single inconsistent result causes r_{10} to be found inconsistent.
- Proposed promises are created “on-demand” to specify the prerequisite of an analysis result. For example, q_1 , q_2 , and q_3 were created to specify the prerequisite of a result. In addition, many “copies” of the same proposed promise may be created, however, this does not occur in this example.

The analysis output for `SynchronizedBoolean` is shown in Figure 2.8. The analysis output after examining `Object` (within the Java standard library) is shown in Figure 2.9.

Definition 2.3.1 is adequate for the formal models we develop in this chapter, however, it is an abstraction of the information collected about each analysis result in the JSure prototype tool. In particular, the tool also collects descriptive text similar to the last column in Figures 2.7, 2.8, and 2.9. The engineering of the JSure prototype tool is described in the next chapter.

Analysis Results for SynchronizedBoolean

	Finding	About	Prerequisite	Description
f_8	+	r_4	$q_4 \vee (q_5 \wedge q_6)$	thread-confined access to <code>value_</code> at line 23
f_9	+	r_{18}	q_7	<code>super()</code> promises consistent effects
f_{10}	+	r_{18}	\top	<code>value_</code> write masked at line 23
f_{11}	+	r_{18}	\top	<code>initialValue</code> read masked at line 23
f_{12}	+	r_{19}	q_8	<code>super()</code> promises it starts no threads
f_{13}	+	r_{19}	\top	constructor starts no threads
f_{14}	+	r_{20}	q_9	<code>super()</code> promises not to alias <code>this</code>
f_{15}	+	r_{20}	\top	constructor does not alias <code>this</code>

Proposed Promises

	Promise	On
q_4	@Unique("return")	SynchronizedBoolean(boolean) at line 21
q_5	@RegionEffects("reads All")	SynchronizedBoolean(boolean) at line 21
q_6	@Starts("nothing")	SynchronizedBoolean(boolean) at line 21
q_7	@RegionEffects("none")	SynchronizedVariable() at line 11
q_8	@Starts("nothing")	SynchronizedVariable() at line 11
q_9	@Unique("return")	SynchronizedVariable() at line 11

Figure 2.8: Analysis output for the SynchronizedBoolean compilation unit in Figure 2.4.

Analysis Results for java.lang.Object

	Finding	About	Prerequisite	Description
f_{16}	+	r_{29}	\top	constructor has consistent effects
f_{17}	+	r_{30}	\top	constructor starts no threads
f_{18}	+	r_{31}	\top	constructor does not alias <code>this</code>

Figure 2.9: Analysis output for java.lang.Object from the Java standard library with respect to the promises in Figure 2.4.

2.3.1 Requirements for program analyses

Our reporting scheme is designed to support a wide variety of modular program analyses. It provides a consistent interface for these analyses to report their verification findings. We do, however, place the following two requirements on all constituent program analyses that exist within our framework:

1. **Sound:** An analysis must report sound results with respect to the semantics of the promise it is verifying. Because any non-trivial program property is statically undecidable, analyses report *conservative* results (as discussed in Section 2.1.2). It is an advantage of our approach that analyses can be “swapped out” over time as more precise analyses are discovered or become computationally tractable on the computers available to tool users.
2. **Prerequisites decoupled from real promises:** An analysis should report a prerequisite assertion in terms of proposed promises. It does not examine the location of a proposed promise to see if the promise exists and then report an inconsistent result if it does not. This remains true even if the proposed promise is within the compilation unit being examined. An example of this is f_8 in Figure 2.8. The prerequisite q_4 proposes a `@Unique("return")` promise on the `SynchronizedBoolean(boolean)` constructor which is within the compilation unit under examination. This proposed promise, in fact, exists as the real promise r_{20} , but the analysis should not use this fact when reporting a prerequisite.

The second requirement is not essential to the soundness of our approach. If an analysis reports a real promise as a prerequisite then any derived verification results are still valid. Proposed promises provide a way for an analysis to precisely specify assertions that, if true, would result in the satisfaction of assertions that the analysis is trying to verify (rather than just produce a textual description for the programmer). Proposed promises also allow tool support for annotating that promise in the programmer’s code at the correct location. The use of proposed promises by analyses, therefore, helps to improve the user experience of tools that use our approach.

We note that this abductive reasoning, where the tool has a “hunch” that a promise is needed in the programmer’s code, is necessarily heuristic. In particular, we do not require constituent analyses to report the “weakest” prerequisites. Thus, a better version of an existing analysis may report identical findings but offer prerequisites that are weaker, in that they admit more models. This has the effect of lessening the requirement to establish that an appropriately placed real promise implies a given proposed promise, since the proposed promise is a weaker assertion. This matching task is called “promise matching” and is described in the next section.

One example where it is expedient to not decouple prerequisites from real promises is the permissions analysis developed by Boyland, Retert, and Zhao [23, 22]. This analysis examines real promises for the purpose of avoiding, in some cases, prerequisites that consist of the disjunction of hundreds of assertions.

Our goal of separation of prerequisites from real promises is similar to the approach taken by Ancona, *et al.* in [5]. They propose a system of compositional compilation for Java that allows the checking of source code fragments in isolation. They produce, for each source code

$$\begin{aligned}
r_{29} &\rightarrow q_1 \\
r_{30} &\rightarrow q_2 \\
r_{31} &\rightarrow q_3 \\
r_{20} &\rightarrow q_4 & (2.1) \\
r_{18} &\rightarrow q_5 & (2.2) \\
r_{19} &\rightarrow q_6 & (2.3) \\
r_8 &\rightarrow q_7 & (2.4) \\
r_9 &\rightarrow q_8 \\
r_{10} &\rightarrow q_9
\end{aligned}$$

Figure 2.10: The elements contained in the set of promise logic formulas, Φ , obtained from promise matching the proposed promises in Figures 2.7, 2.8, and 2.9 to the real promises within the `util.concurrent` code in Figure 2.4.

fragment, a set of type constraints that are resolved during linking. Their type constraints are similar in spirit to our use of proposed promises. Their approach, however, is motivated by a desire to resolve a clash that they see between compilation and the use of dynamic linking in Java (and also C#) while ours is focused on improving the user experience of tools that use our approach.

2.4 Promise matching

After the analysis results and associated proposed promises have been inferred for a set of compilation units, C , containing a set of real promises, P_{real} , we have a set of proposed promises, P_{prop} . Using this information we need to “match” proposed promises to real promises. Each match found is reported as a *promise matching formula*.

Definition 2.4.1 (Promise matching formula). A promise matching formula is a well-formed promise logic formula of the form $r \rightarrow q$ where $r \in \mathbf{RealPromise}$ and $q \in \mathbf{ProposedPromise}$. We define **PMFormula** to be the set of all promise matching formulas.

Promise matching is a specialized program analysis that takes a collection of proposed promises and identifies, for each, a real promise that implies it. Promise matching is a specialized constituent analysis in the sense that it relies on programming language and assertion semantics to complete the matching. It works by examining the location of each proposed promise in the code and checking if an equivalent or stronger real promise exists at that location. That is, a given real promise is only a match if its consistency guarantees the consistency of the proposed promise. The output of promise matching is a set of promise matching formulas, which we refer to as Φ . Promise matching ensures that if $r_1 \rightarrow q \in \Phi$ and $r_2 \rightarrow q \in \Phi$ then $r_1 = r_2$, *i.e.*, a unique result is reported for each consequent.

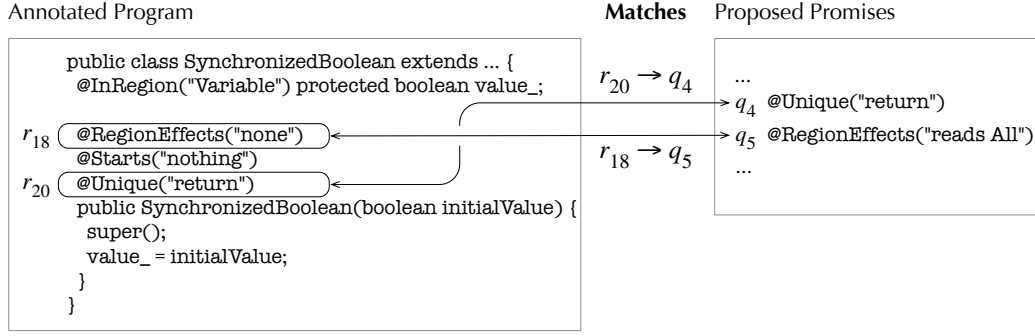


Figure 2.11: An illustration of promise matching for two proposed promises. The proposed promise q_4 exactly matches the real promise r_{20} in the code (in location and semantics). The real promise r_{18} constrains the effects of the same block of code more than the proposed promise q_5 , therefore, r_{18} is a match for q_5 .

For the example `util.concurrent` code in Figure 2.4 the input to promise matching is

$$\begin{aligned}
C &= \{\text{SynchronizedVariable}, \text{SynchronizedBoolean}, \text{Object}\} \\
P_{\text{real}} &= \{r_4, r_8, r_9, r_{10}, r_{18}, r_{19}, r_{20}, r_{29}, r_{30}, r_{31}\} \\
P_{\text{prop}} &= \{q_1, q_2, q_3, \dots, q_9\} \\
R &= \{f_1, f_2, f_3, \dots, f_{18}\}
\end{aligned}$$

and the resulting set of promise matching formulas, Φ , is shown in Figure 2.10. To understand this process better we now examine how two of the promise matching formulas in Figure 2.10 were included in Φ . These two matches are illustrated in Figure 2.11.

Consider q_4 which proposes a `@Unique("return")` promise for the constructor, `SynchronizedBoolean(boolean)`, at line 21. Examination of this constructor finds that an equivalent real promise, r_{20} , exists. Hence, promise matching adds $r_{20} \rightarrow q_4$ to Φ .

As a second example, consider q_5 which proposes a `@RegionEffects("reads All")` for the constructor, `SynchronizedBoolean(boolean)`, at line 21. Examination of this constructor does not find an exactly equivalent promise, however, the effects promised by r_{18} , `@RegionEffects("none")`, are stronger than those proposed by q_5 , therefore, promise matching adds $r_{18} \rightarrow q_5$ to Φ .

2.5 Proof calculus for promise verification

To produce verification proofs that promises are consistent with examined code we introduce a logic to reason about triples of the form

$$\{\psi\} \text{ analysis output } \{\phi\}$$

which roughly means

If the program (that is the object of the analysis output) is run in such a manner that the assertion made by promise logic formula ψ is an invariant, then the assertion made by the promise logic formula ϕ is an invariant.

By *analysis output* we refer to the merged analysis results and promise matching formulas produced by the activities shown in Figure 2.2 and summarized in Figure 2.12 for our running `util.concurrent` example.

Definition 2.5.1 (Promise verification triple). A *promise verification triple* is a triple of the form $\{\psi\}S\{\phi\}$ where ψ is a formula in promise logic called the *prerequisite assertion* and ϕ is a formula in promise logic called the *consequential assertion*. S is a tuple of the form $(\wp(\mathbf{Result}) \times \wp(\mathbf{PMFormula}))$ that contains, as we will see in the examples below, a set of analysis results and a set of promise matching formulas.

There is a crude analogy with the definitions used in the classic verification literature in the tradition of Hoare logic [60]. (See Section 2.8 for an elaboration of the nature of the analogy.) However, it must be emphasized that ψ is *not* a precondition and ϕ is *not* a postcondition—there is no notion of control flow from ψ to ϕ . Traditionally, a precondition, ψ , would be before a block of code, B , and a postcondition, ψ , would be after it; we would say that if B is started in a state that satisfies ψ , then the state after running B will satisfy ϕ [79]. In analysis-based verification, the corresponding terms, prerequisite assertion and consequential assertion, are used with respect to the consistency of promises with the program. The consistency of ψ with the program is a sufficient condition to establish the *consistency* of ϕ with the program.

For example, a `@Borrowed` parameter in a method may force a `@Borrowed` requirement, which is a “consequent invariant,” on parameters of subsidiary methods in a potential call chain. There are dependency relationships among these invariants that are needed to achieve inferential outcomes (*i.e.*, successful proofs) that are only indirectly related to program flow of control. Thus, the precondition precedes along a control-flow path, but the consequent invariant precedes along a proof-structure path. Operationally in the proof process (as noted in the discussion at the end of Section 2.3.1) consequent invariants are inferred in an abductive sense.

Promise verification triples are used to represent and symbolically reason about the consistency of real promises as they relate to each other. For example, using the analysis output from our `util.concurrent` example, the promise verification triple

$$\{q_4 \vee (q_5 \wedge q_6)\}(\{f_8\}, \Phi) \{r_4\}$$

states that the locking model `VarLock`, specified by the `@RegionLock` promise r_4 , is consistent if the `@Unique("return")` promise, q_4 , proposed on the `SynchronizedBoolean(boolean)` constructor at line 21 in Figure 2.4 is consistent or if both the `@RegionEffects("reads All")` promise, q_5 , and the `@Starts("nothing")` promise, q_6 , proposed on the `SynchronizedBoolean(boolean)` constructor are consistent. This triple is supported by one analysis result, f_8 in Figure 2.12, which is the only result for r_4 reported during the analysis of `SynchronizedVariable`, `SynchronizedBoolean`, and `Object`, the set of compilation units examined. The set of formulas produced by promise matching, Φ , is listed in Figure 2.12. In general, the middle portion of the triple tracks the set of analysis results required to support the promise verification triple and the complete set of formulas produced by promise matching.

Analysis Results				
	Finding	About	Prerequisite	Description
f_1	+	r_8	q_1	<code>super()</code> promises consistent effects
f_2	+	r_8	\top	<code>lock_</code> write masked at line 12
f_3	+	r_8	\top	<code>this</code> read masked at line 12
f_4	+	r_9	q_2	<code>super()</code> promises it starts no threads
f_5	+	r_9	\top	constructor starts no threads
f_6	+	r_{10}	q_3	<code>super()</code> promises not to alias <code>this</code>
f_7	\times	r_{10}	\top	<code>this</code> aliased into <code>lock_</code> at line 12
f_8	+	r_4	$q_4 \vee (q_5 \wedge q_6)$	thread-confined access to <code>value_</code> at line 23
f_9	+	r_{18}	q_7	<code>super()</code> promises consistent effects
f_{10}	+	r_{18}	\top	<code>value_</code> write masked at line 23
f_{11}	+	r_{18}	\top	<code>initialValue</code> read masked at line 23
f_{12}	+	r_{19}	q_8	<code>super()</code> promises it starts no threads
f_{13}	+	r_{19}	\top	constructor starts no threads
f_{14}	+	r_{20}	q_9	<code>super()</code> promises not to alias <code>this</code>
f_{15}	+	r_{20}	\top	constructor does not alias <code>this</code>
f_{16}	+	r_{29}	\top	constructor has consistent effects
f_{17}	+	r_{30}	\top	constructor starts no threads
f_{18}	+	r_{31}	\top	constructor does not alias <code>this</code>

Proposed Promises		
	Promise	On
q_1	<code>@RegionEffects("none")</code>	<code>java.lang.Object</code> no-argument constructor
q_2	<code>@Starts("nothing")</code>	<code>java.lang.Object</code> no-argument constructor
q_3	<code>@Unique("return")</code>	<code>java.lang.Object</code> no-argument constructor
q_4	<code>@Unique("return")</code>	<code>SynchronizedBoolean(boolean)</code> at line 21
q_5	<code>@RegionEffects("reads All")</code>	<code>SynchronizedBoolean(boolean)</code> at line 21
q_6	<code>@Starts("nothing")</code>	<code>SynchronizedBoolean(boolean)</code> at line 21
q_7	<code>@RegionEffects("none")</code>	<code>SynchronizedVariable()</code> at line 11
q_8	<code>@Starts("nothing")</code>	<code>SynchronizedVariable()</code> at line 11
q_9	<code>@Unique("return")</code>	<code>SynchronizedVariable()</code> at line 11

Matched Promises (the set Φ)

$r_{29} \rightarrow q_1$
 $r_{30} \rightarrow q_2$
 $r_{31} \rightarrow q_3$
 $r_{20} \rightarrow q_4$
 $r_{18} \rightarrow q_5$
 $r_{19} \rightarrow q_6$
 $r_8 \rightarrow q_7$
 $r_9 \rightarrow q_8$
 $r_{10} \rightarrow q_9$

Figure 2.12: Analysis output for the `util.concurrent` code in Figure 2.4.

$$\begin{aligned}
& \{q_4 \vee (q_5 \wedge q_6)\} (\{f_8\}, \Phi) \{r_4\} \\
& \{q_1 \wedge \top \wedge \top\} (\{f_1, f_2, f_3\}, \Phi) \{r_8\} \\
& \{q_2 \wedge \top\} (\{f_4, f_5\}, \Phi) \{r_9\} \\
& \{q_3 \wedge \top\} (\{f_6, f_7\}, \Phi) \{r_{10}\} \\
& \{q_7 \wedge \top \wedge \top\} (\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\} \\
& \{q_8 \wedge \top\} (\{f_{12}, f_{13}\}, \Phi) \{r_{19}\} \\
& \{q_9 \wedge \top\} (\{f_{14}, f_{15}\}, \Phi) \{r_{20}\} \\
& \{\top\} (\{f_{16}\}, \Phi) \{r_{29}\} \\
& \{\top\} (\{f_{17}\}, \Phi) \{r_{30}\} \\
& \{\top\} (\{f_{18}\}, \Phi) \{r_{31}\}
\end{aligned} \tag{2.5}$$

$$\tag{2.6}$$

Figure 2.13: The elements contained in the set of promise verification conditions, V , obtained from applying Definition 2.5.3 to the analysis output shown in Figure 2.12 about the `util.concurrent` code in Figure 2.4.

2.5.1 Verification condition generation

We now describe how to “bridge the gap” between our analysis and promise matching output to promise verification triples. Intuitively, for each real promise, r , that has only consistent (+) analysis results about it we are able to construct promise verification triples and, from these, develop a set of verification conditions.

Definition 2.5.2 (Promise verification condition). A *promise verification condition* for a promise, r , is the promise verification triple $\text{pvc}(r, R, \Phi)$ where

$$\text{pvc}(r, R, \Phi) = \left\{ \bigwedge_{(+, r, \phi) \in R} \phi \right\} (\{x \in R \mid x = (+, r, \phi)\}, \Phi) \{r\},$$

R is a set of analysis results, and Φ is a set of promise matching formulas.

The definition below uses “ \setminus ” as the set difference operator, *i.e.*, $A \setminus B = \{x : x \in A \wedge x \notin B\}$.

Definition 2.5.3 (Promise verification conditions). The set of *promise verification conditions*, which we call V , is a set of promise verification triples such that $V = \text{vc}(R, \Phi)$ where

$$\begin{aligned}
\text{vc}(R, \Phi) &= \{\text{pvc}(r, R, \Phi) \mid r \in \text{only_consistent_results}(R)\}, \\
\text{only_consistent_results}(R) &= \{r \mid (+, r, \psi) \in R\} \setminus \{r \mid (\times, r, \psi) \in R\},
\end{aligned}$$

R is a set of analysis results, and Φ is a set of promise matching formulas.

These definitions create a promise verification triple for each real promise that has only consistent analysis results reported about it. For example, consider the set of verification conditions, V , shown in Figure 2.13, determined by applying Definition 2.5.3 to the analysis output from our `util.concurrent` example. The promise verification condition for the real promise r_{20} is $\{q_9 \wedge \top\} (\{f_{14}, f_{15}\}, \Phi) \{r_{20}\}$. This triple is an element of V because r_{20} is a real

$$\begin{array}{c}
\frac{\{\phi\}(R, \Phi) \{\psi\} \in V}{V \vdash_{coe} \{\phi\}(R, \Phi) \{\psi\}} \text{Axiom} \\[2ex]
\frac{V \vdash_{coe} \{\phi\}(R_1, \Phi) \{\psi\} \quad V \vdash_{coe} \{\eta\}(R_2, \Phi) \{\theta\}}{V \vdash_{coe} \{\phi \wedge \eta\}(R_1 \cup R_2, \Phi) \{\psi \wedge \theta\}} \text{Merge} \\[2ex]
\frac{V \vdash_{coe} \{\phi \wedge \eta\}(R, \Phi) \{\eta \wedge \psi\}}{V \vdash_{coe} \{\phi\}(R, \Phi) \{\eta \wedge \psi\}} \text{Reduce} \\[2ex]
\frac{\text{seq}(\Phi) \vdash_{pl} \phi' \rightarrow \phi \quad V \vdash_{coe} \{\phi\}(R, \Phi) \{\psi\} \quad \vdash_{pl} \psi \rightarrow \psi'}{V \vdash_{coe} \{\phi'\}(R, \Phi) \{\psi'\}} \text{Implied}
\end{array}$$

A useful derived rule:

$$\frac{V \vdash_{coe} \{\phi\}(R_1, \Phi) \{\eta\} \quad V \vdash_{coe} \{\eta\}(R_2, \Phi) \{\psi\}}{V \vdash_{coe} \{\phi\}(R_1 \cup R_2, \Phi) \{\psi\}} \text{Compose}$$

Figure 2.14: Proof rules for promise verification.

promise and neither of its two results reported an inconsistency (*i.e.*, “×”). The two analysis results for r_{20} , f_{13} and f_{14} , are listed in Figure 2.12 and the conjunction of their prerequisites is the promise logic formula $q_9 \wedge \top$.

There is no promise verification condition in V for r_{10} , the `@Unique("return")` promise on line 10 of Figure 2.4. Definition 2.5.3 excluded this element because one of the two analysis results reported about this promise in Figure 2.12, f_7 , is inconsistent.

2.5.2 Proof rules

The proof rules for the verification of promises are shown in Figure 2.14.

Definition 2.5.4. If Φ is a set of promise logic formulas, we define $\text{seq}(\Phi)$ to be the sequence of formulas derived from the set Φ .

Definition 2.5.5. If the sequent $V \vdash_{coe} \{\phi\}(R, \Phi) \{\psi\}$, where V is a set of promise verification triples, can be derived in the calculus shown in Figure 2.14, then it is valid.

We use a *coe* subscript on \vdash to highlight the intuition that a “chain of evidence” exists from the prerequisite assertion, via the analysis results, to the consequential assertion.

The derived Compose rule

The Compose rule, as demonstrated in Theorem 2.5.1, is a derived rule. The limitation of this classic verification rule [48, 60] in our approach is that it is unable to deal with results obtained from recursive code. Therefore, our calculus defines the Merge and Reduce rules

which do not have this limitation. The **Reduce** rule, in particular, is key to our ability to reason about promises in recursive code.

Theorem 2.5.1 (Compose rule). *Given $V \vdash_{coe} \{\phi\} (R_1, \Phi) \{\eta\}$ and $V \vdash_{coe} \{\eta\} (R_2, \Phi) \{\psi\}$, we can derive, using the proof rules shown in Figure 2.14, $V \vdash_{coe} \{\phi\} (R_1 \cup R_2, \Phi) \{\psi\}$.*

Proof.

$$\frac{\frac{\frac{V \vdash_{coe} \{\phi\} (R_1, \Phi) \{\eta\} \quad V \vdash_{coe} \{\eta\} (R_2, \Phi) \{\psi\}}{V \vdash_{coe} \{\phi \wedge \eta\} (R_1 \cup R_2, \Phi) \{\eta \wedge \psi\}} \text{ Merge}}{\frac{V \vdash_{coe} \{\phi\} (R_1 \cup R_2, \Phi) \{\eta \wedge \psi\}}{V \vdash_{coe} \{\phi\} (R_1 \cup R_2, \Phi) \{\psi\}} \text{ Reduce}} \text{ Implied}$$

□

Before we describe the use of the rules in Figure 2.14 we discuss, informally, the goal of the verification proofs we construct.

2.5.3 The goal

Our goal is to demonstrate that a promise is consistent with the examined code. If that is not possible then we want to be able to understand why it is not possible to demonstrate consistency. Therefore, for each verification condition we want to establish, in order from most desirable to least desirable, a prerequisite assertion of:

1. **The formula \top :** This result indicates model–code consistency. Any other result indicates that the model is incomplete (*i.e.*, for the given set of analyses promises need to be added to the code) or that the code is inconsistent with the model.
2. **A formula containing only proposed promise symbols:** This result indicates that the model is incomplete. The user, to work toward a complete model, must add additional promises to the code. In this case, and the two below, we assume that any proposed promise symbols in the prerequisite assertion that could be removed have been removed (*i.e.*, the remaining proposed promises were not “matched”).
3. **A formula containing both proposed and real promise symbols:** The proposed promise symbols in the prerequisite assertion indicate that the model is incomplete (*i.e.*, promises need to be added to the code) as described above. Further, the real promise symbols in the prerequisite assertion indicate that either an inconsistent analysis result was reported for those promises or they were not checked by an analysis as described below.
4. **A formula containing only real promise symbols:** This result indicate that for each real promise in the prerequisite assertion either (a) the analysis results contained an inconsistent report about that promise or (b) no analysis checked its consistency. Our formal model does not distinguish between these two cases. However, an examination of the analysis results makes it possible to distinguish between them and this is the approach used by the JSure prototype tool (as described in the next chapter).

This heuristic approach, other than a preference for a prerequisite assertion of \top , may seem odd. Why not just fail to produce a result if model-code consistency cannot be demonstrated? The heuristic above is designed to make the results as helpful to the user as possible, with a goal of helping the user to understand how to take a “next step” toward consistency.

In the following sections, using our example from `util.concurrent`, we describe the use of the rules in Figure 2.14.

2.5.4 Including verification conditions

The Axiom rule

$$\frac{\{\phi\}(R, \Phi) \{\psi\} \in V}{V \vdash_{coe} \{\phi\}(R, \Phi) \{\psi\}} \text{Axiom}$$

allows us to derive members of V as part of our proof. For example, consider the promise verification condition in Equation 2.6 about the `@RegionEffects("none")` promise, r_{18} , at line 18 in Figure 2.4,

$$\{q_7 \wedge \top \wedge \top\}(\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}$$

using the Axiom rule we can derive

$$\overline{V \vdash_{coe} \{q_7 \wedge \top \wedge \top\}(\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}} \text{Axiom}$$

because $\{q_7 \wedge \top \wedge \top\}(\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\} \in V$.

2.5.5 Linking to promise logic

The Implied rule

$$\frac{\text{seq}(\Phi) \vdash_{pl} \phi' \rightarrow \phi \quad V \vdash_{coe} \{\phi\}(R, \Phi) \{\psi\} \quad \vdash_{pl} \psi \rightarrow \psi'}{V \vdash_{coe} \{\phi'\}(R, \Phi) \{\psi'\}} \text{Implied}$$

forms a “link” between our verification logic and our promise logic. This rule allows the prerequisite assertion to be strengthened and the consequential assertion to be weakened. This rule is adopted from Hoare [60] where it linked program logic to first-order predicate logic augmented with basic facts about arithmetic. The primary purpose of the `Implied` rule in our calculus is to incorporate the results of promise matching into the prerequisite assertion of a promise verification triple. It allows us to “replace” proposed promises with real promises using the set of formulas produced by promise matching, Φ .

The `Implied` rule allows any formula in $\text{seq}(\Phi)$, the sequence of formulas derived from the set Φ , to be used as a premise during the strengthening of a prerequisite assertion. For example, consider the promise verification condition in Equation 2.6 about the `@RegionEffects("none")` promise, r_{18} , at line 18 in Figure 2.4,

$$\{q_7 \wedge \top \wedge \top\}(\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}$$

and the formula, $r_8 \rightarrow q_7$, in Equation 2.4 which is an element of Φ (identified during promise matching). Because

$$r_8 \rightarrow q_7 \vdash_{pl} r_8 \rightarrow q_7 \wedge \top \wedge \top$$

is a valid sequent in promise logic, we can derive

$$\frac{\overline{V \vdash_{coe} \{q_7 \wedge \top \wedge \top\} (\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}}}{V \vdash_{coe} \{r_8\} (\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}} \begin{array}{l} \text{Axiom} \\ \text{Implied} \end{array}$$

which simplifies the prerequisite assertion of our triple and replaces the proposed promise q_7 with the real promise r_8 . This replacement of a proposed promise with a real promise sets up our triple about r_{18} to be composed (using the **Compose** rule) with a triple about r_8 .

Our general approach to replace proposed promises with real promises is to examine a prerequisite formula, ϕ , and within that formula “replace” each matched proposed promise with its real promise to produce a new formula, ψ . We then must derive $\text{seq}(\Phi) \vdash_{pl} \psi \rightarrow \phi$ to allow use of **Implied**.

As a more complex example, consider the promise verification condition in Equation 2.5 about the **@RegionLock** promise, r_4 , at line 4 in Figure 2.4,

$$\{q_4 \vee (q_5 \wedge q_6)\} (\{f_8\}, \Phi) \{r_4\}$$

and the formulas $r_{20} \rightarrow q_4$, $r_{18} \rightarrow q_5$, and $r_{19} \rightarrow q_6$ from Equations 2.1, 2.2, and 2.3 which are elements of Φ . Because

$$r_{20} \rightarrow q_4, r_{18} \rightarrow q_5, r_{19} \rightarrow q_6 \vdash_{pl} r_{20} \vee (r_{18} \wedge r_{19}) \rightarrow q_4 \vee (q_5 \wedge q_6)$$

is a valid sequent in promise logic, we can derive

$$\frac{\overline{V \vdash_{coe} \{q_4 \vee (q_5 \wedge q_6)\} (\{f_8\}, \Phi) \{r_4\}}}{V \vdash_{coe} \{r_{20} \vee (r_{18} \wedge r_{19})\} (\{f_8\}, \Phi) \{r_4\}} \begin{array}{l} \text{Axiom} \\ \text{Implied} \end{array}$$

which is now free of proposed promises.

2.5.6 Composing results

The rules presented above allow us to introduce verification conditions into proofs and to replace proposed promises with real promises within the prerequisite assertions of our triples. We now discuss the rules that allow us to compose our triples into a larger verification proof.

Notice that the `util.concurrent` code we have been using as a running example in this chapter does not contain recursion. Recursion within the program can result in triples where a promise directly or indirectly requires itself as a prerequisite. We address recursion below in Section 2.5.7.

The **Compose** rule

$$\frac{V \vdash_{coe} \{\phi\} (R_1, \Phi) \{\eta\} \quad V \vdash_{coe} \{\eta\} (R_2, \Phi) \{\psi\}}{V \vdash_{coe} \{\phi\} (R_1 \cup R_2, \Phi) \{\psi\}} \text{Compose}$$

allows us to compose triples and work toward a prerequisite assertion of \top . This rule used with the **Axiom** and **Implied** rules is enough to prove the consistency of the **@Starts("nothing")** promise, r_{19} , on line 19 in Figure 2.4 with the code. This proof, shown in Figure 2.15, demonstrates that r_{19} is consistent by first replacing all proposed promises in the prerequisite assertions then using the **Compose** rule to derive a prerequisite assertion of \top . The

$$\begin{array}{c}
\frac{\overline{V \vdash_{coe} \{\top\} (\{f_{17}\}, \Phi) \{r_{30}\}} \text{ Axiom} \quad \frac{\overline{V \vdash_{coe} \{q_2 \wedge \top\} (\{f_4, f_5\}, \Phi) \{r_9\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_{30}\} (\{f_4, f_5\}, \Phi) \{r_9\}} \text{ Implied (1)}} \text{ Compose} \\
\hline
V \vdash_{coe} \{\top\} (\{f_4, f_5, f_{17}\}, \Phi) \{r_9\}
\end{array}$$

$$\begin{array}{c}
\frac{\text{above } \nearrow \quad \overline{V \vdash_{coe} \{\top\} (\{f_4, f_5, f_{17}\}, \Phi) \{r_9\}} \quad \frac{\overline{V \vdash_{coe} \{q_8 \wedge \top\} (\{f_{12}, f_{13}\}, \Phi) \{r_{19}\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_9\} (\{f_{12}, f_{13}\}, \Phi) \{r_{19}\}} \text{ Implied (2)}} \text{ Compose} \\
\hline
V \vdash_{coe} \{\top\} (\{f_4, f_5, f_{12}, f_{13}, f_{17}\}, \Phi) \{r_{19}\}
\end{array}$$

Figure 2.15: Proof of the sequent $V \vdash_{coe} \{\top\} (\{f_4, f_5, f_{12}, f_{13}, f_{17}\}, \Phi) \{r_{19}\}$ which demonstrates the consistency of the `@Starts("nothing")` promise, r_{20} , on line 20 in Figure 2.4 with the code. The valid sequent used by **Implied (1)** is $r_{30} \rightarrow q_2 \vdash_{pl} r_{30} \rightarrow q_2 \wedge \top$ and by **Implied (2)** is $r_9 \rightarrow q_8 \vdash_{pl} r_9 \rightarrow q_8 \wedge \top$. V is defined in Figure 2.13.

$$\begin{array}{c}
\frac{\overline{V \vdash_{coe} \{\top\} (\{f_{16}\}, \Phi) \{r_{29}\}} \text{ Axiom} \quad \frac{\overline{V \vdash_{coe} \{q_1 \wedge \top \wedge \top\} (\{f_1, f_2, f_3\}, \Phi) \{r_8\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_{29}\} (\{f_1, f_2, f_3\}, \Phi) \{r_8\}} \text{ Implied (1)}} \text{ Compose} \\
\hline
V \vdash_{coe} \{\top\} (\{f_1, f_2, f_3, f_{16}\}, \Phi) \{r_8\}
\end{array}$$

$$\begin{array}{c}
\frac{\text{above } \nearrow \quad \overline{V \vdash_{coe} \{\top\} (\{f_1, f_2, f_3, f_{16}\}, \Phi) \{r_8\}} \quad \frac{\overline{V \vdash_{coe} \{q_7 \wedge \top \wedge \top\} (\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{18}\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_8\} (\{f_9, f_{10}, f_{11}\}, \Phi) \{r_{19}\}} \text{ Implied (2)}} \text{ Compose} \\
\hline
V \vdash_{coe} \{\top\} (\{f_1, f_2, f_3, f_9, f_{10}, f_{11}, f_{16}\}, \Phi) \{r_{19}\}
\end{array}$$

Figure 2.16: Proof of the sequent $V \vdash_{coe} \{\top\} (\{f_1, f_2, f_3, f_9, f_{10}, f_{11}, f_{16}\}, \Phi) \{r_{18}\}$ which demonstrates the consistency of the `@RegionEffects("none")` promise, r_{18} , on line 18 in Figure 2.4 with the code. The valid sequent used by **Implied (1)** is $r_{29} \rightarrow q_1 \vdash_{pl} r_{29} \rightarrow q_1 \wedge \top \wedge \top$ and by **Implied (2)** is $r_8 \rightarrow q_7 \vdash_{pl} r_8 \rightarrow q_7 \wedge \top \wedge \top$. V is defined in Figure 2.13.

structure of this proof illustrates that `@Starts("nothing")` promise, r_{30} , is satisfied on the `java.lang.Object` no-argument constructor, which is required for `@Starts("nothing")` promise, r_9 , to be satisfied on the `SynchronizedVariable` no-argument constructor, which is required for `@Starts("nothing")` promise, r_{19} , to be satisfied on the `Synchronized-Boolean(boolean)` constructor.

The proof in Figure 2.16 demonstrates the consistency of the `@RegionEffects("none")` promise, r_{18} , on line 18 in Figure 2.4 with the code and is similar in approach to the proof described above.

To prove the `VarLock` model is consistent with the code we need to demonstrate the consistency of the `@RegionLock` promise, r_4 . This proof is shown in Figure 2.17 and it requires use of the **Merge** rule

$$\frac{V \vdash_{coe} \{\phi\} (R_1, \Phi) \{\psi\} \quad V \vdash_{coe} \{\eta\} (R_2, \Phi) \{\theta\}}{V \vdash_{coe} \{\phi \wedge \eta\} (R_1 \cup R_2, \Phi) \{\psi \wedge \theta\}} \text{ Merge}$$

to create the promise logic formula $r_{18} \wedge r_{19}$ and allow use of the **Compose** rule. Note the use

$$\begin{aligned}
R_1 &= \{f_1, f_2, f_3, f_9, f_{10}, f_{11}, f_{16}\} \\
R_2 &= \{f_4, f_5, f_{12}, f_{13}, f_{17}\} \\
R_3 &= R_1 \cup R_2 \\
R_4 &= R_3 \cup \{f_8\}
\end{aligned}$$

Figure 2.16	Figure 2.15			
$V \vdash_{coe} \{\top\} (R_1, \Phi) \{r_{18}\}$	$V \vdash_{coe} \{\top\} (R_2, \Phi) \{r_{19}\}$	Merge	$V \vdash_{coe} \{q_4 \vee (q_5 \wedge q_6)\} (\{f_8\}, \Phi) \{r_4\}$	Axiom
$V \vdash_{coe} \{\top \wedge \top\} (R_3, \Phi) \{r_{18} \wedge r_{19}\}$			$V \vdash_{coe} \{r_{20} \vee (r_{18} \wedge r_{19})\} (\{f_8\}, \Phi) \{r_4\}$	Implied
$V \vdash_{coe} \{\top\} (R_3, \Phi) \{r_{18} \wedge r_{19}\}$		Implied	$V \vdash_{coe} \{r_{18} \wedge r_{19}\} (\{f_8\}, \Phi) \{r_4\}$	Implied
$V \vdash_{coe} \{\top\} (R_4, \Phi) \{r_4\}$			Compose	

Figure 2.17: Proof of the sequent $V \vdash_{coe} \{\top\} (R_4, \Phi) \{r_4\}$ which demonstrates the consistency of the `@RegionLock("VarLock is lock_ protects Variable")` promise, r_4 , on line 4 in Figure 2.4 with the code. V is defined in Figure 2.13.

of the **Implies** rule to choose the right-hand side of $r_{20} \vee (r_{18} \wedge r_{29})$ because we can derive a prerequisite assertion of \top for $r_{18} \wedge r_{29}$ while the best we can do for r_{20} is derive the sequent

$$V \vdash_{coe} \{r_{10}\} (\{f_{14}, f_{15}\}, \Phi) \{r_{20}\}$$

which means that the `@Unique("return")` promise, r_{20} on on the `SynchronizedBoolean(boolean)` constructor is consistent if the `@Unique("return")` promise, r_{10} on on the `Synchronized-Variable` constructor is consistent. However, as discussed in Section 2.5.3, a prerequisite assertion of \top is considered a better result than a prerequisite assertion of r_{10} so we choose that path in our proof of of the `@RegionLock` promise, r_4 . In addition, the uniqueness analysis reported inconsistent results about r_{10} .

2.5.7 Handling recursion

Recursion within the program can result in triples where a promise directly or indirectly requires itself as a prerequisite. This comes about because a method, directly or indirectly, calls itself. The **Reduce** rule

$$\frac{V \vdash_{coe} \{\phi \wedge \eta\} (R, \Phi) \{\eta \wedge \psi\}}{V \vdash_{coe} \{\phi\} (R, \Phi) \{\eta \wedge \psi\}} \text{ Reduce}$$

allows us to “remove the loop” from a proof. It captures the necessary coinductive argument needed to handle promises about recursive methods. Using coinduction, in a sense, we assume the desired result in order to prove it. This sounds dangerous, however, with the restrictions we have placed on our formalisms, it results in valid proofs. Coinduction, as a proof technique for program analysis, is discussed in Appendix B of [87]. The need for the **Reduce** rule to handle verification of recursive code is the driver behind much of the complexity (*e.g.*, needing both promise logic, \vdash_{pl} , and our verification calculus, \vdash_{coe}) of the formalisms presented in this chapter. It also motivates the soundness proof presented in Section 2.7 because it makes this property non-obvious.

```

1 public class Fibonacci {
2     long callsToFibMethod = 0;
3
4     @Borrowed("this")
5     public long fib(int n) {
6         callsToFibMethod++;
7         if (n <= 1)
8             return n;
9         else
10            return fib(n-1) + fib(n-2);
11     }
12 }

```

Analysis Results for Fibonacci

	Finding	About	Prerequisite	Description
f_1	+	r_4	q_1	<code>fib(n-1)</code> at line 10 promises not to alias <code>this</code>
f_2	+	r_4	q_2	<code>fib(n-2)</code> at line 10 promises not to alias <code>this</code>
f_3	+	r_4	\top	<code>fib(int)</code> does not alias <code>this</code>

Proposed Promises

	Promise	On
q_1	<code>@Borrowed("this")</code>	<code>fib(int)</code> at line 5
q_2	<code>@Borrowed("this")</code>	<code>fib(int)</code> at line 5

$$\begin{aligned}
C &= \{\text{Fibonacci}\} \\
R &= \{f_1, f_2, f_3\} \\
\Phi &= \{r_4 \rightarrow q_1, r_4 \rightarrow q_2\} \\
V &= \{\{q_1 \wedge q_2 \wedge \top\} (R, \Phi) \{r_4\}\}
\end{aligned}$$

Proof of the sequent $V \vdash_{coe} \{\top\} (R, \Phi) \{r_4\}$

$$\begin{array}{c}
\frac{}{V \vdash_{coe} \{q_1 \wedge q_2 \wedge \top\} (R, \Phi) \{r_4\}} \text{Axiom} \\
\frac{}{V \vdash_{coe} \{\top \wedge r_4\} (R, \Phi) \{r_4 \wedge \top\}} \text{Implied} \\
\frac{}{V \vdash_{coe} \{\top\} (R, \Phi) \{r_4 \wedge \top\}} \text{Reduce} \\
\frac{}{V \vdash_{coe} \{\top\} (R, \Phi) \{r_4\}} \text{Implied}
\end{array}$$

Figure 2.18: An example of using the Reduce rule to handle recursion in a promise verification proof. (Top) The Java code for (inefficiently) computing a Fibonacci number using recursive calls. The `Fibonacci` instance tracks the number of calls to `fib`. The `fib` method promises that it does not alias the receiver. (Middle) Analysis results, proposed promises, matched promises, and verification conditions. (Bottom) The proof of the sequent $V \vdash_{coe} \{\top\} (R, \Phi) \{r_4\}$ which demonstrates the consistency of the `@Borrowed("this")` promise, r_4 , on line 4 with the code.


```

1 public class Recursive {
2   @Borrowed("this")
3   public Recursive() {
4     go(null);
5   }
6
7   @Borrowed("this")
8   void go(@Borrowed Object work) {
9     if (work == null)
10      stop(this);
11   }
12
13   @Borrowed("this")
14   void stop(@Borrowed Object done) {
15     go(done);
16   }
17 }

```

Analysis Results for Recursive

	Finding	About	Prerequisite	Description
f_1	+	r_2	\top	constructor does not alias this
f_2	+	r_2	q_1	<code>go(null)</code> at line 4 promises not to alias this
f_3	+	r_7	\top	<code>go(Object)</code> does not alias this
f_4	+	r_7	q_2	<code>stop(this)</code> at line 10 promises not to alias this
f_5	+	r_7	q_3	<code>stop(this)</code> at line 10 promises not to alias argument 1
f_6	+	r_8	\top	<code>go(Object)</code> does not alias argument 1, <i>i.e.</i> , work
f_7	+	r_{13}	\top	<code>stop(Object)</code> does not alias this
f_8	+	r_{13}	q_4	<code>go(done)</code> at line 15 promises not to alias this
f_9	+	r_{14}	\top	<code>stop(Object)</code> does not alias argument 1, <i>i.e.</i> , done
f_{10}	+	r_{14}	q_5	<code>go(done)</code> at line 15 promises not to alias argument 1

Proposed Promises

	Promise	On
q_1	@Borrowed("this")	<code>go(Object)</code> at line 8
q_2	@Borrowed("this")	<code>stop(Object)</code> at line 14
q_3	@Borrowed	argument 1 of <code>stop(Object)</code> at line 14
q_4	@Borrowed("this")	<code>go(Object)</code> at line 8
q_5	@Borrowed	argument 1 of <code>go(Object)</code> at line 8

Figure 2.19: Code and analysis results for a contrived class with mutually recursive methods.

$$\begin{aligned}
C &= \{\text{Recursive}\} \\
R &= \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\} \\
\Phi &= \{r_7 \rightarrow q_1, r_{13} \rightarrow q_2, r_{14} \rightarrow q_3, r_7 \rightarrow q_4, r_8 \rightarrow q_5\} \\
V &= \{ \\
&\quad \{\top \wedge q_1\} (\{f_1, f_2\}, \Phi) \{r_2\}, \\
&\quad \{\top \wedge q_2 \wedge q_3\} (\{f_3, f_4, f_5\}, \Phi) \{r_7\}, \\
&\quad \{\top\} (\{f_6\}, \Phi) \{r_8\}, \\
&\quad \{\top \wedge q_4\} (\{f_7, f_8\}, \Phi) \{r_{13}\}, \\
&\quad \{\top \wedge q_5\} (\{f_9, f_{10}\}, \Phi) \{r_{14}\}, \\
&\quad \}
\end{aligned}$$

$$\frac{\overline{V \vdash_{coe} \{\top\} (\{f_6\}, \Phi) \{r_8\}} \text{ Axiom} \quad \frac{\overline{V \vdash_{coe} \{\top \wedge q_5\} (\{f_9, f_{10}\}, \Phi) \{r_{14}\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_8\} (\{f_9, f_{10}\}, \Phi) \{r_{14}\}} \text{ Implied}} \text{ Compose}$$

$$\frac{\overline{V \vdash_{coe} \{\top\} (\{f_6, f_9, f_{10}\}, \Phi) \{r_{14}\}} \text{ above } \nearrow \quad \frac{\overline{V \vdash_{coe} \{\top \wedge q_4\} (\{f_7, f_8\}, \Phi) \{r_{13}\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_7\} (\{f_7, f_8\}, \Phi) \{r_{13}\}} \text{ Implied}} \text{ Merge}$$

$$\frac{\overline{V \vdash_{coe} \{\top \wedge r_7\} (\{f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_{14} \wedge r_{13}\}} \text{ above } \nearrow \quad \frac{\overline{V \vdash_{coe} \{\top \wedge q_2 \wedge q_3\} (\{f_3, f_4, f_5\}, \Phi) \{r_7\}} \text{ Axiom}}{\overline{V \vdash_{coe} \{r_{14} \wedge r_{13}\} (\{f_3, f_4, f_5\}, \Phi) \{r_7\}} \text{ Implied}} \text{ Compose}$$

$$\frac{\overline{V \vdash_{coe} \{\top \wedge r_7\} (\{f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_7\}} \text{ Implied}}{\overline{V \vdash_{coe} \{\top \wedge r_7\} (\{f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_7 \wedge \top\}} \text{ Reduce}} \text{ Implied}$$

$$\frac{\overline{V \vdash_{coe} \{\top\} (\{f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_7 \wedge \top\}} \text{ Implied}}{\overline{V \vdash_{coe} \{\top\} (\{f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_7\}} \text{ Implied}}$$

Figure 2.20: An example of using the **Reduce** rule to handle mutual recursion in a promise verification proof. (Top) Compilation units, analysis results, matched promises, and verification conditions. (Bottom) Proof of the sequent $V \vdash_{coe} \{\top\} (\{f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}, \Phi) \{r_7\}$ which demonstrates the consistency of the `@Borrowed("this")` promise, r_7 , on line 7 in Figure 2.19 with the code.

$$\begin{aligned}
C &= \{\text{TravelAgentBean}\} \\
R &= \{f_1, f_2, f_3, f_4, f_5, f_6\} \\
\Phi &= \{r_{12} \rightarrow q_1\} \\
V &= \{ \\
&\quad \{\top \wedge q_1 \wedge q_2 \wedge q_3\} (\{f_1, f_2, f_3, f_4\}, \Phi) \{r_6\}, \\
&\quad \{\top \wedge q_4\} (\{f_5, f_6\}, \Phi) \{r_{12}\} \\
&\quad \}
\end{aligned}$$

$$\begin{array}{c}
\frac{V \vdash_{coe} \{\top \wedge q_4\} (\{f_5, f_6\}, \Phi) \{r_{12}\}}{V \vdash_{coe} \{q_4\} (\{f_5, f_6\}, \Phi) \{r_{12}\}} \text{Axiom} \\
\frac{V \vdash_{coe} \{q_4\} (\{f_5, f_6\}, \Phi) \{r_{12}\}}{V \vdash_{coe} \{q_4 \wedge q_2 \wedge q_3 \wedge r_{12}\} (R, \Phi) \{r_{12} \wedge r_6\}} \text{Implied (1)} \\
\frac{V \vdash_{coe} \{\top \wedge q_1 \wedge q_2 \wedge q_3\} (\{f_1, f_2, f_3, f_4\}, \Phi) \{r_6\}}{V \vdash_{coe} \{q_2 \wedge q_3 \wedge r_{12}\} (\{f_1, f_2, f_3, f_4\}, \Phi) \{r_6\}} \text{Axiom} \\
\frac{V \vdash_{coe} \{q_2 \wedge q_3 \wedge r_{12}\} (\{f_1, f_2, f_3, f_4\}, \Phi) \{r_6\}}{V \vdash_{coe} \{q_4 \wedge q_2 \wedge q_3\} (R, \Phi) \{r_{12} \wedge r_6\}} \text{Implied (2)} \\
\frac{V \vdash_{coe} \{q_4 \wedge q_2 \wedge q_3\} (R, \Phi) \{r_{12} \wedge r_6\}}{V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}} \text{Merge} \\
\frac{V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}}{V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}} \text{Reduce} \\
\frac{V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}}{V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}} \text{Implied (3)}
\end{array}$$

Figure 2.21: Proof of the sequent $V \vdash_{coe} \{q_2 \wedge q_3 \wedge q_4\} (R, \Phi) \{r_6\}$ which demonstrates the consistency of the `@Starts("nothing")` promise, r_6 , on line 6 in Figure 2.1 with the code. For this example, the analysis results and proposed promises are shown in Figure 2.3. The valid sequent used by **Implied** (1) is $\vdash_{pl} q_4 \rightarrow \top \wedge q_4$, by **Implied** (2) is $r_{12} \rightarrow q_1 \vdash_{pl} q_2 \wedge q_3 \wedge r_{12} \rightarrow \top \wedge q_1 \wedge q_2 \wedge q_3$, and by **Implied** (3) is $\vdash_{pl} r_{12} \wedge r_6 \rightarrow r_6$.

For non-recursive code we typically use the simpler **Compose** rule, however, as was demonstrated in the proof of Theorem 2.5.1, this rule is derived from the **Reduce** rule.

Figure 2.18 provides an straightforward example of the **Reduce** rule to verify a `@Borrowed("this")` promise about a recursive method that computes a Fibonacci number. The `@Borrowed("this")` promise asserts that the method will not create any aliases to the receiver, *i.e.*, `this` (as described in A.3.1).

A more complex recursive example is shown in Figure 2.19. This highly-contrived example highlights the flow-sensitive nature of alias promises. The `@Borrowed("this")` promise, r_7 , on the `go` method requires that the method `stop` to promise not to alias its receiver or its first argument because `this` is passed as the first argument of the call to `stop` at line 10. The proof in Figure 2.20 demonstrates that r_7 is consistent with the code in Figure 2.19. This proof demonstrates the use of **Merge** and **Reduce** rules to verify a set of mutually recursive methods.

2.5.8 Handling “mixed” prerequisite assertions

The **Merge** and **Reduce** rules are also needed to produce proofs that have a prerequisite assertion that is composed of a mixture of real and proposed promises. An example of this type of proof is shown in Figure 2.21 about the `TravelAgentBean` example that was used as a motivating example at the beginning of this chapter. In this example the prerequisite assertion for r_6 is derived to be $q_2 \wedge q_3 \wedge r_{12}$ (below **Implied** (2)). This mixture of real and

proposed promises is not in a form that the **Compose** rule can handle. The **Compose** rule can only be used when the prerequisite assertion contains only real promise symbols. Therefore, the **TravelAgentBean** proof uses the more flexible **Merge** and **Reduce** rules to continue the proof and derive the more desirable, as described in Section 2.5.3, prerequisite assertion of $q_2 \wedge q_3 \wedge q_4$.

2.6 Semantics

In this section we develop a precise semantics for promise logic and for analysis results. We first define a truth table semantics for promise logic.

Definition 2.6.1 (Promise logic semantics). The set of values contains two elements T and U, where T represents ‘true’ or ‘consistent’ and U represents ‘unknown’. The truth tables for promise logic match the truth tables for propositional logic (except that U replaces F).

A *model* of a promise logic formula ϕ is an assignment of each atom in ϕ to a truth value. $\mathcal{M}_{(T)}$ is used to denote a model where the set of atoms contained in the set T are assigned T and all other atoms are assigned U.

We say $\mathcal{M}_{(T)} \models_{pl} \psi$ if the promise logic formula ψ evaluates to T by the model $\mathcal{M}_{(T)}$. We say $\mathcal{M}_{(T)} \not\models_{pl} \psi$ if the promise logic formula ψ evaluates to U by the model $\mathcal{M}_{(T)}$.

For example, using this definition we can determine that $\mathcal{M}_{(\{r_1, r_2\})} \models_{pl} (r_1 \wedge r_2) \vee r_3$ and $\mathcal{M}_{(\{r_1\})} \not\models_{pl} r_1 \rightarrow r_2$. Based upon Definition 2.6.1, we can now develop a semantics of analysis results in terms of promise consistency.

Definition 2.6.2 (Analysis result implication). An *analysis result implication* for a promise, r , is the promise logic formula $\text{result_impl}(r, R)$ where

$$\text{result_impl}(r, R) = \left(\bigwedge_{(+, r, \phi) \in R} \phi \right) \rightarrow r$$

and R is a set of analysis results.

Definition 2.6.3 (Analysis implications). The set of *analysis implications*, which we call Ψ , for a set of analysis results, R , and a set of promise matching formulas, Φ , is a set of promise logic formulas such that $\Psi = \text{analysis_impls}(R, \Phi)$ where

$$\begin{aligned} \text{analysis_impls}(R, \Phi) &= \{\text{result_impl}(r, R) \mid r \in \text{only_consistent_results}(R)\} \cup \Phi, \\ \text{only_consistent_results}(R) &= \{r \mid (+, r, \psi) \in R\} \setminus \{r \mid (\times, r, \psi) \in R\}, \end{aligned}$$

R is a set of analysis results, and Φ is a set of promise matching formulas.

Applying these two definitions to the **TravelAgentBean** analysis results shown in Figure 2.3 and Φ at the top of Figure 2.21 results in the following set of analysis implications:

$$\Psi = \{\top \wedge q_1 \wedge q_2 \wedge q_3 \rightarrow r_6, \top \wedge q_4 \rightarrow r_{12}, r_{12} \rightarrow q_1\}$$

As another example, applying these two definitions to the **Recursive** analysis results shown in Figure 2.19 and Φ at the top of Figure 2.20 results in the following set of analysis

implications:

$$\begin{aligned} \Psi = \{ & \top \wedge q_1 \rightarrow r_2, \top \wedge q_2 \wedge q_3 \rightarrow r_7, \top \rightarrow r_8, \top \wedge q_4 \rightarrow r_{13}, \top \wedge q_5 \rightarrow r_{14}, \\ & r_7 \rightarrow q_1, r_{13} \rightarrow q_2, r_{14} \rightarrow q_3, r_7 \rightarrow q_4, r_8 \rightarrow q_5 \} \end{aligned}$$

Note that in both examples, the consequent of each analysis implication is unique. We prove that this useful property holds in general below. In addition, we prove that the antecedent of each analysis implication is implication-free. In the statement of the propositions below we use the term “produced by analysis-based verification of a program” to indicate that the set of analysis results and promise matching formulas were produced by a system that respects the requirements described in Section 2.3.1 and Section 2.4, respectively.

Lemma 2.6.1 (Uniqueness of analysis implication consequents). *For a set of analysis implications, Ψ , produced by analysis-based verification of a program, if $\phi \rightarrow p \in \Psi$ and $\psi \rightarrow p \in \Psi$ then $\phi = \psi$.*

Proof. If p is a proposed promise then $\phi \rightarrow p$ and $\psi \rightarrow p$ must be elements of Φ (the output of promise matching) by examination of how Definition 2.6.3 constructs Ψ . In this case, the proposition holds because promise matching in an analysis-based verification system ensures a unique result is reported for each consequent (*i.e.*, $\phi = \psi$) as described in Section 2.4. If p is a real promise then by examination of how Definition 2.6.3 constructs Ψ , only a single formula has p as its consequent and the proposition holds in this case as well. Therefore, because each promise must be either a proposed promise or a real promise the proposition is true. \square

Lemma 2.6.2 (Analysis implication antecedents are implication-free). *For a set of analysis implications, Ψ , produced by analysis-based verification of a program, if $\phi \rightarrow p \in \Psi$ then $\phi \in \mathbf{AOFormula}$.*

Proof. If p is a proposed promise then $\phi \rightarrow p$ must be an element of Φ (the output of promise matching) by examination of how Definition 2.6.3 constructs Ψ . In this case, the proposition holds because promise matching in an analysis-based verification system guarantees that ϕ takes the form r where r is a real promise as described in Section 2.4 and r is clearly an element of $\mathbf{AOFormula}$. If p is a real promise then by examination of how Definition 2.6.3 constructs Ψ it is a conjunction of implication-free formulas reported as the prerequisite of an analysis result and the proposition holds in this case as well (because conjoining elements of $\mathbf{AOFormula}$ creates a formula that is still an element of $\mathbf{AOFormula}$). Therefore, because each promise must be either a proposed promise or a real promise the proposition is true. \square

Our approach is to formally define the semantics of analysis results in terms of promise consistency with the program (or set of compilation units examined). If a promise, p , is consistent with the program then $p \in T$. If a promise is inconsistent or is contingently consistent with the program then $p \notin T$.

Definition 2.6.4 (Analysis semantics). An *analysis semantics* is the largest set T of assumed \top atoms that is consistent with a set of analysis result implications, Ψ . We say that T is consistent with Ψ if

$$\forall p \in T, \exists (\psi \rightarrow p) \in \Psi : \mathcal{M}_{(T)} \models_{pl} \psi.$$

Applying this definition to the **TravelAgentBean** example,

$$T = \emptyset$$

because all the results are contingent, *i.e.*, their prerequisite assertion is not \top , and, therefore, they are all ‘unknown’.

Applying this definition to the **Recursive** example,

$$T = \{q_1, q_2, q_3, q_4, q_5, r_2, r_7, r_8, r_{13}, r_{14}\}$$

because all the results are consistent with the program, *i.e.*, their prerequisite assertion is \top , and, therefore, they are all ‘true’ or ‘consistent’. Notice that T includes proposed promises that have been matched with a consistent real promise.

We conclude this section by proving a theorem (and a supporting lemma) that a “largest set T ” exists. We do this because it could be considered plausible that there exist two incomparable sets T_1 and T_2 that are consistent with Ψ but that $T_1 \cup T_2$ is not consistent with Ψ .

Lemma 2.6.3 (Monotonicity of implication-free formulas). *Let T_1 and T_2 be sets of assumed T atoms such that $T_1 \neq T_2$. if $\mathcal{M}_{(T_1)} \models_{pl} \phi$ where $\phi \in \mathbf{AOF}\mathbf{ormula}$ then $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \phi$.*

Proof. By induction on the form (structure) of the implication-free promise logic formula ϕ .

(Case p) Where where $p \in \mathbf{Promise}$, *i.e.*, it is a promise symbol. We assume $\mathcal{M}_{(T_1)} \models_{pl} p$ or this case is vacuously true. Using this fact, by Definition 2.6.1, we know that $p \in T_1$. Therefore, $p \in T_1 \cup T_2$, and $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \phi$ and this case holds.

(Case \top) Because $\mathcal{M}_{(T)} \models_{pl} \top$ for any set T of assumed \top atoms (including the empty set), this case is immediate.

(Case $\rho \wedge \chi$) We assume $\mathcal{M}_{(T_1)} \models_{pl} \rho \wedge \chi$ or this case is vacuously true. By the semantics of \wedge we know $\mathcal{M}_{(T_1)} \models_{pl} \rho$ and $\mathcal{M}_{(T_1)} \models_{pl} \chi$, and, thus, by the induction hypothesis $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \rho$ and $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \chi$. Therefore, we can conclude, by the semantics of \wedge , $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \rho \wedge \chi$ and this case holds.

(Case $\rho \vee \chi$) We assume $\mathcal{M}_{(T_1)} \models_{pl} \rho \vee \chi$ or this case is vacuously true. By the semantics of \vee we know that $\mathcal{M}_{(T_1)} \models_{pl} \rho$ or $\mathcal{M}_{(T_1)} \models_{pl} \chi$ or both. We assume, without loss of generality, that $\mathcal{M}_{(T_1)} \models_{pl} \rho$, and, thus, by the induction hypothesis $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \rho$. Therefore, we can conclude, by the semantics of \vee , $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \rho \vee \chi$ and this case holds.

All cases hold so the proposition is true. □

Theorem 2.6.4 (Existence of analysis semantics). *For a set of analysis implications, Ψ , produced by analysis-based verification of a program, a largest set, T , of analysis semantics exists.*

Proof. (By contradiction) We use Definition 2.6.4 to define consistency with Ψ . Assume, to the contrary, that there exists a set T_1 of assumed **T** atoms that is consistent with Ψ and that there exists a set T_2 of assumed **T** atoms that is consistent with Ψ such that $T_1 \neq T_2$ but that $T_1 \cup T_2$ is not consistent with Ψ .

If $T_1 \cup T_2$ is not consistent with Ψ then there exists a $p \in T_1 \cup T_2$ such that either $\psi \rightarrow p \in \Psi$ and $\mathcal{M}_{(T_1 \cup T_2)} \not\models_{pl} \psi$ or $\psi \rightarrow p \notin \Psi$. We know that p is an element of T_1 or T_2 or both. Because both T_1 and T_2 are consistent with Ψ we assume, without loss of generality, that $p \in T_1$ and consider the two cases defined at the start of this paragraph.

(Case $\psi \rightarrow p \in \Psi$ and $\mathcal{M}_{(T_1 \cup T_2)} \not\models_{pl} \psi$) In this case because $p \in T_1$ and T_1 is consistent with Ψ we know by Definition 2.6.4 that there exists $\psi' \rightarrow p \in \Psi$ such that $\mathcal{M}_{(T_1)} \models_{pl} \psi'$. By Lemma 2.6.1 (Uniqueness of analysis implication consequents) we know that $\psi = \psi'$ and, therefore, $\mathcal{M}_{(T_1)} \models_{pl} \psi$. Because $\psi \rightarrow p \in \Psi$, by Lemma 2.6.2 (Analysis implication antecedents are implication-free), we know that $\psi \in \mathbf{AOFormula}$. Further, because $\mathcal{M}_{(T_1)} \models_{pl} \psi$, $T_1 \neq T_2$, and $\psi \in \mathbf{AOFormula}$, by Lemma 2.6.3 (Monotonicity of implication-free formulas), we know $\mathcal{M}_{(T_1 \cup T_2)} \models_{pl} \psi$. However, by the definition of this case $\mathcal{M}_{(T_1 \cup T_2)} \not\models_{pl} \psi$ is a contradiction.

(Case $\psi \rightarrow p \notin \Psi$) In this case because $p \in T_1$ and T_1 is consistent with Ψ we know by Definition 2.6.4 that there exists an implication in Ψ with p as its consequent. However, by the definition of this case this existence is a contradiction.

In both cases we reach a contradiction, therefore, the proposition is true. \square

Note that Theorem 2.6.4 (Existence of analysis semantics), depends upon the prerequisite assertions reported by program analyses as part of each analysis result to be implication-free. If implications are allowed then this theorem is not true. For example, consider the (illegal) set of analysis results $R = \{(+, r_3, q_1 \rightarrow r_2), (+, r_2, \top)\}$ where Φ is empty. By Definition 2.6.3 $\Psi = \{(q_1 \rightarrow r_2) \rightarrow r_3, \top \rightarrow r_2\}$. Let $T_1 = \{r_3\}$ and $T_2 = \{r_2\}$. Notice that, by Definition 2.6.4, T_1 and T_2 are consistent with Ψ but $T_1 \cup T_2$ is not. Avoiding this situation is the reason why the prerequisite for each analysis result must be an element of **AOFormula**.

2.7 Soundness

In this section we prove a soundness theorem that relates the logic for analysis-based verification presented in Section 2.5 to the analysis semantics developed in Section 2.6. Theorem 2.7.4 (Soundness) proves the proposition

$$\text{If } V \vdash_{coe} \{\phi\} (R', \Phi) \{\psi\} \text{ is valid then } \mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi \text{ holds}$$

which states that if we can derive a triple where ϕ is the prerequisite assertion for ψ in our logic then it must be the case that the formula $\phi \rightarrow \psi$ is ‘true’ or ‘consistent’ in our analysis semantics (*i.e.*, $\phi \rightarrow \psi$ evaluates to **T** in the model $\mathcal{M}_{(T)}$). We present proofs for three supporting lemmas followed by a proof of soundness.

First, however, we sketch our overall proof strategy. Lemma 2.7.1 (Consistency of promise matching formulas) asserts that all promise logic formulas in Φ are ‘true’ or ‘consistent’ in our analysis semantics (*i.e.*, each formula in Φ evaluates to **T** in the model $\mathcal{M}_{(T)}$). This lemma is required to prove the soundness of the **Implied** rule in our calculus. The proof for this lemma is straightforward.

The second and third lemma provide an invariant required to prove the soundness of the **Reduce** rule in our calculus.

$$\frac{V \vdash_{coe} \{\phi \wedge \eta\} (R_1, \Phi) \{\eta \wedge \psi\}}{V \vdash_{coe} \{\phi\} (R_1, \Phi) \{\eta \wedge \psi\}} \text{ Reduce}$$

If we examine this rule it is clear that if ϕ is ‘true’ then η must be ‘true’ as well otherwise the **Reduce** rule (in terms of our semantics) turns $(\mathbf{T} \wedge \mathbf{U}) \rightarrow \mathbf{U}$ (which is sound) into $\mathbf{T} \rightarrow \mathbf{U}$ (which is unsound). Lemma 2.7.2 (Coinductive assertions invariant) is the general form of this invariant and its formal proposition is somewhat baroque. Lemma 2.7.3 (Reduce rule invariant) is a more obvious statement of this invariant for the **Reduce** rule and its proof is a straightforward application of Lemma 2.7.2.

The proofs in this section rely upon the fact that promise logic is sound (for a proof see [64]). As in the previous section, several of the propositions below use the term “produced by analysis-based verification of a program” to indicate that the set of analysis results and promise matching formulas were produced by a system that respects the requirements described in Section 2.3.1 and Section 2.4, respectively.

Lemma 2.7.1 (Consistency of promise matching formulas). *Let Φ be a set of promise matching formulas and T be the analysis semantics produced by analysis-based verification of a program. For all promise matching formulas $r \rightarrow q \in \Phi$, $\mathcal{M}_{(T)} \models_{pl} r \rightarrow q$.*

Proof. Let Ψ be the set of analysis implications used to produce T . We consider two cases based upon the semantics of r .

(Case $\mathcal{M}_{(T)} \models_{pl} r$) In this case $r \in T$. By Lemma 2.6.1 (Uniqueness of analysis implication consequents) we know that $r \rightarrow q$ is the only implication in Ψ with q as its consequent. Because $r \in T$ and $r \rightarrow q \in \Psi$ and by Theorem 2.6.4 (Existence of analysis semantics) T is maximal, we know by Definition 2.6.4 that $q \in T$. Therefore, $\mathcal{M}_{(T)} \models_{pl} r \rightarrow q$ and this case holds.

(Case $\mathcal{M}_{(T)} \not\models_{pl} r$) In this case we know, by the semantics of \rightarrow , that $\mathcal{M}_{(T)} \models_{pl} r \rightarrow q$ and this case holds.

In both cases $\mathcal{M}_{(T)} \models_{pl} r \rightarrow q$ so the proposition is true. \square

Lemma 2.7.2 (Coinductive assertions invariant). *Let ϕ' , ψ' , ϕ'' , ψ'' , α , and η be promise logic formulas. Let R be a set of analysis results, Φ a set of promise matching formulas, V a set of promise verification conditions, and T the analysis semantics produced by analysis-based verification of a program. Let $R' \subseteq R$ and $R'' \subseteq R$. If $V \vdash_{coe} \{\phi'\} (R', \Phi) \{\psi'\}$ and $V \vdash_{coe} \{\phi''\} (R'', \Phi) \{\psi''\}$ such that $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi' \wedge \phi''$ and $\vdash_{pl} \psi' \wedge \psi'' \rightarrow \eta$ and $\mathcal{M}_{(T)} \models_{pl} \alpha$ then $\mathcal{M}_{(T)} \models_{pl} \eta$.*

Proof. By structural induction on the proof of $V \vdash_{coe} \{\phi\}(R_1, \Phi) \{\psi\}$ where $R_1 \subseteq R$. We proceed by cases on the final rule used in the derivation of $V \vdash_{coe} \{\phi\}(R_1, \Phi) \{\psi\}$ (the Axiom case is the basis).

(Case Axiom) In this case let $\phi = \phi' = \phi''$, $\psi = \psi' = \psi''$, and $R_1 = R' = R''$. We know, by the Axiom rule, that $\{\phi\}(R_1, \Phi) \{\psi\} \in V$ and, by Definition 2.5.3, that ψ must have the form r where r is a real promise. There exist two promise logic formulas α and η such that $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi$ and $\vdash_{pl} r \rightarrow \eta$. We assume $\mathcal{M}_{(T)} \models_{pl} \alpha$ or this case is vacuously true. By examination of the proof rules for promise logic (in particular “ \rightarrow i” and “ \top i”) it is clear that if $\vdash_{pl} r \rightarrow \eta$ is valid then η must be either r or \top . We now demonstrate $\mathcal{M}_{(T)} \models_{pl} \eta$ in these two cases.

(Case η is \top) This case holds because for any T , $\mathcal{M}_{(T)} \models_{pl} \top$.

(Case η is r) In this case $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge r \rightarrow \phi$ and $\vdash_{pl} r \rightarrow r$. Because $\{\phi\}(R_1, \Phi) \{r\} \in V$ we know that $\phi \rightarrow r \in \Psi$ (due to the similar techniques used to construct V and Ψ by Definition 2.5.3 and Definition 2.6.3, respectively). Lemma 2.7.1 (Consistency of promise matching formulas) tell us that all the all the formulas in Φ are consistent (i.e., $\forall \chi \in \Phi : \mathcal{M}_{(T)} \models_{pl} \chi$), therefore, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge r \rightarrow \phi$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge r \rightarrow \phi$). Further, by our assumption, we know that α is consistent ($\mathcal{M}_{(T)} \models_{pl} \alpha$), therefore, only r could be keeping r out of T (because if r is consistent then ϕ must be consistent and, by Definition 2.6.4, $\phi \rightarrow r \in \Psi$ would ensure r is an element of the largest set T) which by Definition 2.6.4 and Theorem 2.6.4 (Existence of analysis semantics) is not possible— r cannot keep itself out of T . Therefore, $\mathcal{M}_{(T)} \models_{pl} r$ and this case holds.

(Case Merge) In this case let $\phi = \phi' = \phi''$, $\psi = \psi' = \psi''$, and $R_1 = R' = R''$. We know, by the Merge rule, that ϕ is of the form $\tau \wedge v$ and that ψ is of the form $\sigma \wedge \rho$. There exist two promise logic formulas α and η such that $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \tau \wedge v$ and $\vdash_{pl} \sigma \wedge \rho \rightarrow \eta$. We assume $\mathcal{M}_{(T)} \models_{pl} \alpha$ or this case is vacuously true. We need to demonstrate $\mathcal{M}_{(T)} \models_{pl} \eta$.

In this rule above the bar we have two sequents $V \vdash_{coe} \{\tau\}(R_2, \Phi) \{\sigma\}$ and $V \vdash_{coe} \{v\}(R_3, \Phi) \{\rho\}$ where $R_1 = R_2 \cup R_3$. By the induction hypothesis, we know $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \tau \wedge v$ and $\vdash_{pl} \sigma \wedge \rho \rightarrow \eta'$, $\mathcal{M}_{(T)} \models_{pl} \alpha'$, and $\mathcal{M}_{(T)} \models_{pl} \eta'$.

Lemma 2.7.1 (Consistency of promise matching formulas) tell us that all the all the formulas in Φ are consistent (i.e., $\forall \chi \in \Phi : \mathcal{M}_{(T)} \models_{pl} \chi$), therefore, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \tau \wedge v$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \tau \wedge v$). Similarly, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \tau \wedge v$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \tau \wedge v$). Further, because $\mathcal{M}_{(T)} \models_{pl} \alpha'$, $\mathcal{M}_{(T)} \models_{pl} \eta'$, and $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \tau \wedge v$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \tau \wedge v$.

Therefore, because $\mathcal{M}_{(T)} \models_{pl} \alpha$, $\mathcal{M}_{(T)} \models_{pl} \tau \wedge v$, and $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \tau \wedge v$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \eta$ and this case holds.

(Case Reduce) In this case let $\phi = \phi' = \phi''$, $\psi = \psi' = \psi''$, and $R_1 = R' = R''$. We know, by the Reduce rule, that ψ is of the form $\sigma \wedge \rho$. There exist two promise logic formulas α and η such that $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi$ and $\vdash_{pl} \sigma \wedge \rho \rightarrow \eta$. We assume $\mathcal{M}_{(T)} \models_{pl} \alpha$ or this case is vacuously true. We need to demonstrate $\mathcal{M}_{(T)} \models_{pl} \eta$.

In this rule above the bar we have the sequent $V \vdash_{coe} \{\phi \wedge \sigma\} (R_1, \Phi) \{\sigma \wedge \rho\}$. Using this sequent twice, by the induction hypothesis, we know $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \phi \wedge \sigma$ and $\vdash_{pl} \sigma \wedge \rho \rightarrow \eta'$, $\mathcal{M}_{(T)} \models_{pl} \alpha'$, and $\mathcal{M}_{(T)} \models_{pl} \eta'$.

Lemma 2.7.1 (Consistency of promise matching formulas) tell us that all the all the formulas in Φ are consistent (*i.e.*, $\forall \chi \in \Phi : \mathcal{M}_{(T)} \models_{pl} \chi$), therefore, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \phi \wedge \sigma$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \phi \wedge \sigma$). Similarly, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \phi$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi$). Further, because $\mathcal{M}_{(T)} \models_{pl} \alpha'$, $\mathcal{M}_{(T)} \models_{pl} \eta'$, and $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \phi \wedge \sigma$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \phi \wedge \sigma$ and, by the semantics of \wedge , $\mathcal{M}_{(T)} \models_{pl} \phi$.

Therefore, because $\mathcal{M}_{(T)} \models_{pl} \alpha$, $\mathcal{M}_{(T)} \models_{pl} \phi$, and $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \phi$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \eta$ and this case holds.

(Case Implied) In this case let $\phi = \phi' = \phi''$, $\psi = \psi' = \psi''$, and $R_1 = R' = R''$. There exist two promise logic formulas α and η such that $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi$ and $\vdash_{pl} \psi \rightarrow \eta$. We assume $\mathcal{M}_{(T)} \models_{pl} \alpha$ or this case is vacuously true. We need to demonstrate $\mathcal{M}_{(T)} \models_{pl} \eta$.

In this rule above the bar we have the sequent $V \vdash_{coe} \{\xi\} (R_1, \Phi) \{\gamma\}$ where $\text{seq}(\Phi) \vdash_{pl} \phi \rightarrow \xi$ and $\vdash_{pl} \psi \rightarrow \gamma$. Using this sequent twice, by the induction hypothesis, we know $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \xi$ and $\vdash_{pl} \gamma \rightarrow \eta'$, $\mathcal{M}_{(T)} \models_{pl} \alpha'$, and $\mathcal{M}_{(T)} \models_{pl} \eta'$.

Lemma 2.7.1 (Consistency of promise matching formulas) tell us that all the all the formulas in Φ are consistent (*i.e.*, $\forall \chi \in \Phi : \mathcal{M}_{(T)} \models_{pl} \chi$), therefore, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \xi$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha' \wedge \eta' \rightarrow \xi$). Similarly, we know that $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \phi$ (because $\text{seq}(\Phi) \vdash_{pl} \alpha \wedge \eta \rightarrow \phi$) and $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \xi$ (because $\text{seq}(\Phi) \vdash_{pl} \phi \rightarrow \xi$). Further, because $\mathcal{M}_{(T)} \models_{pl} \alpha'$, $\mathcal{M}_{(T)} \models_{pl} \eta'$, and $\mathcal{M}_{(T)} \models_{pl} \alpha' \wedge \eta' \rightarrow \xi$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \xi$. Similarly, because $\mathcal{M}_{(T)} \models_{pl} \xi$ and $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \xi$, by the semantics of \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \phi$.

Therefore, because $\mathcal{M}_{(T)} \models_{pl} \alpha$, $\mathcal{M}_{(T)} \models_{pl} \phi$, and $\mathcal{M}_{(T)} \models_{pl} \alpha \wedge \eta \rightarrow \phi$, by the semantics of \wedge and \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \eta$ and this case holds.

The basis and all cases hold so the proposition is true. \square

Lemma 2.7.3 (Reduce rule invariant). *Let ϕ , η , and ψ be promise logic formulas. Let R be a set of analysis results, Φ a set of promise matching formulas, V a set of promise verification conditions, and T be the analysis semantics produced by analysis-based verification of a program. Let $R' \subseteq R$. If $V \vdash_{coe} \{\phi \wedge \eta\} (R', \Phi) \{\eta \wedge \psi\}$ and $\mathcal{M}_{(T)} \models_{pl} \phi$ then $\mathcal{M}_{(T)} \models_{pl} \eta$.*

Proof. We proceed by application of Lemma 2.7.2 (Coinductive assertions invariant). Using $V \vdash_{coe} \{\phi \wedge \eta\} (R', \Phi) \{\eta \wedge \psi\}$ for both sequents in the hypothesis of Lemma 2.7.2, it is clear that $\text{seq}(\Phi) \vdash_{pl} \phi \wedge \eta \rightarrow (\phi \wedge \eta) \wedge (\phi \wedge \eta)$ and $\vdash_{pl} (\eta \wedge \psi) \wedge (\eta \wedge \psi) \rightarrow \eta$ are both valid. By the proposition (of this lemma) we know $\mathcal{M}_{(T)} \models_{pl} \phi$, therefore, by Lemma 2.7.2, we conclude $\mathcal{M}_{(T)} \models_{pl} \eta$ and the proposition is true. \square

Theorem 2.7.4 (Soundness). *Let ϕ and ψ be promise logic formulas. Let R be a set of analysis results, Φ a set of promise matching formulas, V a set of promise verification conditions, and T be the analysis semantics produced by analysis-based verification of a program. Let $R' \subseteq R$. If $V \vdash_{coe} \{\phi\} (R', \Phi) \{\psi\}$ is valid then $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi$ holds.*

Proof. By structural induction on the proof of $V \vdash_{coe} \{\phi\} (R', \Phi) \{\psi\}$. We proceed by cases on the final rule used in the derivation of $V \vdash_{coe} \{\phi\} (R', \Phi) \{\psi\}$ demonstrating that in each case $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi$ holds (the Axiom case is the basis).

(Case Axiom) We know, by the Axiom rule, that $\{\phi\} (R', \Phi) \{\psi\} \in V$ and, by Definition 2.5.3, that ψ must have the form r where r is a real promise. Therefore, we must demonstrate that $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow r$ holds. Let Ψ be the set of analysis implications used to produce T . Because $\{\phi\} (R', \Phi) \{r\} \in V$ we know that $\phi \rightarrow r \in \Psi$ (due to the similar techniques used to construct V and Ψ by Definition 2.5.3 and Definition 2.6.3, respectively). Further, we know, by Lemma 2.6.1 (Uniqueness of analysis implication consequents) that $\phi \rightarrow r$ is the only formula in Ψ with r as the conclusion of the implication. We consider two cases based upon the semantic value of r .

(Case $\mathcal{M}_{(T)} \models_{pl} r$) In this case, by Definition 2.6.1, we know that $r \in T$. Because $\phi \rightarrow r$ is the only formula in Ψ with r as the conclusion of the implication, by Definition 2.6.4, $\mathcal{M}_{(T)} \models_{pl} \phi$ must hold or r could not be an element of T . Therefore, we can conclude $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow r$ and this case holds.

(Case $\mathcal{M}_{(T)} \not\models_{pl} r$) In this case, by Definition 2.6.1, we know that $r \notin T$. Because $\phi \rightarrow r$ is the only formula in Ψ with r as the conclusion of the implication, by Definition 2.6.4, $\mathcal{M}_{(T)} \not\models_{pl} \phi$ must be true or r would be an element of T . Therefore, we can conclude $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow r$ and this case holds.

(Case Merge) In this case we know that ϕ is of the form $\phi' \wedge \eta$ and that ψ is of the form $\psi' \wedge \theta$. By the induction hypothesis we know $\mathcal{M}_{(T)} \models_{pl} \phi' \rightarrow \psi'$ and $\mathcal{M}_{(T)} \models_{pl} \eta \rightarrow \theta$. Therefore, by the semantics of promise logic, we can conclude $\mathcal{M}_{(T)} \models_{pl} \phi' \wedge \eta \rightarrow \psi' \wedge \theta$ and this case holds.

(Case Reduce) In this case we know that ψ is of the form $\eta \wedge \psi'$. By the induction hypothesis we know $\mathcal{M}_{(T)} \models_{pl} \phi \wedge \eta \rightarrow \eta \wedge \psi'$. We consider two cases based upon the semantic value of $\phi \wedge \eta$.

(Case $\mathcal{M}_{(T)} \models_{pl} \phi \wedge \eta$) In this case we know $\mathcal{M}_{(T)} \models_{pl} \eta \wedge \psi'$ by the induction hypothesis and the semantics of \rightarrow . We also know $\mathcal{M}_{(T)} \models_{pl} \phi$ by the semantics of \wedge . Therefore, we can conclude $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \eta \wedge \psi'$ and this case holds.

(Case $\mathcal{M}_{(T)} \not\models_{pl} \phi \wedge \eta$) In this case we know, by Lemma 2.7.3 (Reduce rule invariant), that $\mathcal{M}_{(T)} \not\models_{pl} \phi$ because if $\mathcal{M}_{(T)} \models_{pl} \phi$ then $\mathcal{M}_{(T)} \models_{pl} \eta$ (which would imply $\mathcal{M}_{(T)} \models_{pl} \phi \wedge \eta$ and contradict this case). Therefore, by the semantics of \rightarrow , $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \eta \wedge \psi'$ and this case holds.

(Case Implied) In this case, there exist two promise logic formulas ϕ' and ψ' such that $\text{seq}(\Phi) \vdash_{pl} \phi \rightarrow \phi'$ and $\vdash_{pl} \psi \rightarrow \psi'$. Further, by the induction hypothesis, we know $\mathcal{M}_{(T)} \models_{pl} \phi' \rightarrow \psi'$. We consider two cases based upon the semantic value of ϕ .

(Case $\mathcal{M}_{(T)} \models_{pl} \phi$) Lemma 2.7.1 (Consistency of promise matching formulas) tell us that all the all the formulas in Φ are consistent (*i.e.*, $\forall \chi \in \Phi : \mathcal{M}_{(T)} \models_{pl} \chi$), therefore, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \phi'$ (because $\text{seq}(\Phi) \vdash_{pl} \phi \rightarrow \phi'$). Further, because $\mathcal{M}_{(T)} \models_{pl} \phi$ and $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \phi'$, by the semantics of \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \phi'$. Similarly, because $\mathcal{M}_{(T)} \models_{pl} \phi'$ and $\mathcal{M}_{(T)} \models_{pl} \phi' \rightarrow \psi'$, we know $\mathcal{M}_{(T)} \models_{pl} \psi'$. Because $\vdash_{pl} \psi' \rightarrow \psi$ is valid, by the soundness of promise logic, we know that $\mathcal{M}_{(T)} \models_{pl} \psi' \rightarrow \psi$. Using this fact and $\mathcal{M}_{(T)} \models_{pl} \psi'$, by the semantics of \rightarrow , we know $\mathcal{M}_{(T)} \models_{pl} \psi$. Therefore, we can conclude $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi$ and this case holds.

(Case $\mathcal{M}_{(T)} \not\models_{pl} \phi$) In this case we can conclude that $\mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi$ holds by the semantics of \rightarrow .

The basis and all cases hold so the proposition is true. □

2.8 Not a Hoare logic

Although we use a notation reminiscent of Hoare [60], our calculus for promise verification would not, traditionally, be considered a Hoare logic for the following reasons:

- The proof rules defined by a Hoare logic define semantics for a programming language, referred to as an axiomatic semantics. Our framework does not directly or indirectly define semantics of a programming language. This is embodied, in part, in the constituent analyses.
- As discussed in Section 2.5, the prerequisite assertion and consequential assertion of our triples represent assertions about the consistency of promises and not, as within a Hoare logic, assertions about the state of a program. Promises may, in fact, represent an assertion about the state of the program, however, that fact is not visible to our framework.
- A Hoare logic is used to prove partial or total correctness of a program. Our approach cannot prove partial or total correctness of a program. The assertions made by promises typically focus on mechanical properties rather than the functionality of the program.
- For any interesting programming language, proving assertions (typically made in first-order propositional logic enlarged with basic facts about arithmetic) using a Hoare logic, is not fully automatable. Our approach is, by design, automatable.

Modern work developing Hoare logics, such as [1] for object-oriented programs, use a much different formal approach than the one presented in this chapter for the reasons listed above.

2.9 Conclusion

This chapter presents a formal model that describes the construction of proofs within an analysis-based verification system. This includes specifying formally how constituent program analyses report their findings and an automatable proof calculus to create program- or component-level results based upon these findings. We present a soundness theorem that relates the proof calculus to a semantics of analysis results.

Our approach to analysis-based verification supports separate analysis of components and allows composition of the results such that the outcome corresponds to that of a whole-program analysis. The requirement to support modularity and composability permeate the formal systems presented in this chapter. The well-defined separation between analysis and proof management (*i.e.*, result merging, promise matching, and the construction of verification proofs) provide the foundation for the tool engineering we present in the next chapter.

Realization

“In the computer field, the moment of truth is running a program; all else is prophecy.”
— Herbert Simon

3.1 Introduction

In this chapter we build upon the formal model developed in the previous chapter and present significant details about the design and engineering of the JSure prototype analysis-based verification tool. The realization of the JSure tool has evolved based upon feedback from its use in several field trials. These field trials are presented in Chapter 4. Chapter 1 sketched, in its tour of analysis-based verification, most of the topics presented in this chapter. This chapter, however, provides a more complete presentation with significantly more technical detail.

We provide partial solutions to two problems that arise when realizing a practical extra-linguistic verification system within a modern IDE: (1) representing and managing verification results in such a way that a programmer can understand them as changes are made to both code and models, and (2) expressing design intent about large software systems containing multiple components developed by separate teams. The requirement to overcome barriers to *adoption* and issues of *scale* permeates the technical solutions presented in this chapter.

The approach we take to implementing sound combined analyses for analysis-based verification yields the following technical and engineering results (repeated from Chapter 1) which are summarized below and elaborated in the remainder of this chapter.

- **The drop-sea proof management system:** Drop-sea is the proof management system used by the JSure tool. Drop-sea manages the results reported by constituent program analyses and automates the proof calculus presented in the previous chapter to create verification results based upon these findings.
- **Management of contingencies—the red dot:** Drop-sea allows several unverified contingencies to exist in a chain of evidence about a promise. A programmer can vouch for an overly conservative analysis result—changing it from an “x” to a “+”. A programmer can turn off a particular program analysis causing all the promises checked by that analysis to have no results—causing the tool to trust these promises without any

analysis evidence. Finally, the programmer can assume something about a component that is outside of the programmer’s scope of interest (*e.g.*, on the other side of an organizational or contractual boundary). These actions introduce a contingency into any proof that relies upon them. Drop-sea explicitly tracks these contingencies and flags them with a red dot.

- **Proposed promises:** Our approach has constituent analyses report any necessary prerequisite assertions as part of each analysis result. Analyses, when they report a prerequisite assertion, propose promises that may or may not exist in the code. A special analysis called *promise matching* is used to “match” each proposed promise with a programmer-expressed promise in the code. If no “match” can be found, *i.e.*, a promise proposed by a constituent analysis is not in the code base, then the computation that produces verification results is able to use the unmatched proposed promises to determine the “weakest” prerequisite assertion for each promise in the code base. This allows the tool to propose “missing” annotations, from the point of view of the constituent analyses, to the code that can be reviewed and accepted by the tool user.
- **Scoped promises:** Scoped promises are promises that act on other promises or analysis results within a static scope of code. We introduce three types of scoped promises: `@Promise` to avoid repetitive user annotation of the same promise over and over again in a class or package, `@Assume` to support team modeling in large systems where programmers are not permitted access to the entire system’s code, and `@Vouch` to quiet overly conservative analysis results. Scoped promises help to “scale up” the ability of a programmer or a team of programmers to express design intent about a large software system.

A principal contribution of this thesis is combining all of the items listed above in a new way to facilitate extra-linguistic verification of real-world software systems. (This contribution was referred to as “user experience design and tool engineering approach” in Section 1.6.2.)

The integration of the JSure tool into the Eclipse Java IDE and the development of a graphical user interface allowed us to package the prototype tool in a widely-adopted environment that many programmers are very comfortable with. According to a 2007 Forrester research study [59] Eclipse is used by 57% of Java programmers—more than all other Java IDEs put together. The familiarity of Eclipse and the user interface of the prototype tool have helped to enable the field trials that we present in the next chapter. However, we view the current user interface of the tool to be utilitarian—it enables the tool user to interact with JSure and accomplish necessary tasks, but it is still far from providing an optimal or elegant user experience. We delay our (and outside tool users) critique of the JSure tool user interface until the next chapter when we discuss the feedback from our field trials.

To understand how a programmer uses the JSure prototype tool, the next section presents an introduction to the programmer–tool interaction. We then present an overview of the tool architecture. We discuss each of the tool’s major components and how they interact. The next section provides further detail on drop-sea, a key contribution of our work, and how it accomplishes its role of proof management and truth maintenance for the system. We then describe the three types of *scoped promises* supported by the JSure tool: `@Promise` to avoid repetitive user annotation of the same promise over and over again in a class or package, `@Assume` to support team modeling in large systems where programmers are not permitted access to the entire system’s code, and `@Vouch` to quiet overly conservative analysis results.

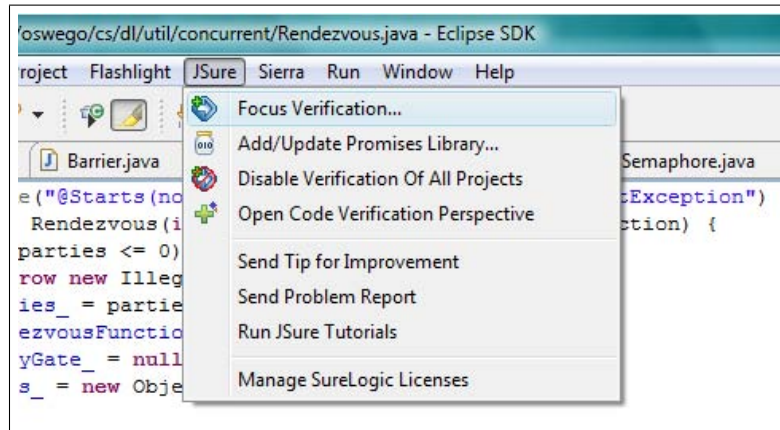


Figure 3.1: Focusing the JSure tool on a particular Java project of interest.

We then describe how we handle *trusted promises*, assertions that are not verified by analysis, and the tool’s support for them. We end the chapter with a discussion of related work.

3.2 Programmer-tool interaction

From the JSure menu the programmer “turns on” the tool for a particular Java project using the Focus Verification... menu choice as shown in Figure 3.1. To “turn off” the JSure tool the programmer selects the Disable Verification Of All Projects menu choice.

This action enables the JSure tool and adds a library (named `promises.jar`) containing the promise annotations to the build path of the selected project.

3.2.1 Tool suggestions to get started

To help the programmer get started, even with no annotations yet in the code, the tool displays a list of suggestions intended to highlight locations in the code where annotations could be placed. The rationale for this feature is to help support the *incremental reward principle* (Section 1.6.1). An example of these tool suggestions is shown in Figure 3.2. The user is given a large number of locations where threads and locks are defined or used within the code. The items indicated with a blue “i” are informational messages that may help the programmer understand some aspect of the code. For example, the “4 java.lang.Thread subtype instance creation(s)” may be worth examining as the programmer works to understand where threads are created and started in the code.

The items indicated with the yellow warning symbol “⚠” highlight code that may need to be annotated or changed. For example, the “505 unidentifiable lock(s); what is the name of the lock? what state is being protected?” warning points to locations in the code where a lock is being acquired/released. This warning identifies locks that could have the state that they are intended to protect expressed with a `@RegionLock` or `@GuardedBy` annotation. The “8 non-final lock expression(s); analysis cannot determine which lock is being acquired” warning indicates places in the code where a field referencing an object used as a lock appears to be mutable. It is good practice to use the `final` keyword to ensure that the same lock object

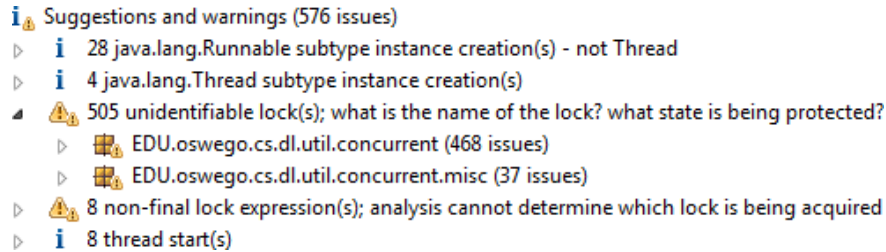


Figure 3.2: Suggestions on where to start annotating are shown in the *Verification Status* view.

is always referenced. Using these warnings the programmer can fix the declarations of the indicated fields.

3.2.2 Adding annotations

The programmer enters promise annotations into the code using the Eclipse Java editor. The tool provides help on the syntax and semantics of the annotations used to express design intent in the tool. For example, the declaration of each annotation contains significant use documentation (in Javadoc) that “pops up” in the Java editor as shown in Figure 3.3. The tool provides an on-line version of Appendix A as well as several tutorials that introduce the annotation language and present typical examples of its use.

The tool also, through a facility provided by the Eclipse IDE, supports several templates to assist the tool user with the annotation syntax. Each template adds one or more annotations to the code and allows the user to enter any missing information using a fill-in-the-blank style. An example of using a template to enter the `@RegionLock` promise to the `BoundedFIFO` class (used as a running example in Chapter 1) is shown in Figure 3.4. This seemingly straightforward feature, as will be discussed in Chapter 4, helped to enable client programmers that participated in our field trials to enter annotations without assistance from us. Prior to the inclusion of this feature, we typed most of the annotations into the code. After its inclusion, the programmers “typed” most of the annotations themselves using templates.

The rationale for providing “pop up” Javadoc and syntax templates is to help support model expression in JSure.

3.2.3 Tool output

As soon as the programmer saves a Java file containing even a single annotation, the JSure tool performs a build, executes analyses, and displays its output. An example of the tool displaying its output to the user is shown in Figure 3.5. JSure defines a new Eclipse perspective—a task-oriented arrangement of windows within the IDE—that is called ‘Code Verification’ to organize the presentation of the tool results for the user. Verification results for annotations that are well-formed are shown in the *Verification Status* view. The iconography used in this view and how to interpret the verification results are discussed, in detail, throughout the remainder of this chapter. A guide to this iconography is provided, for reference, in Figure 3.6.

A list of proposed promises, representing missing portions of models that were proposed by the program analyses, is shown in the *Proposed Promises* view. The user can add these

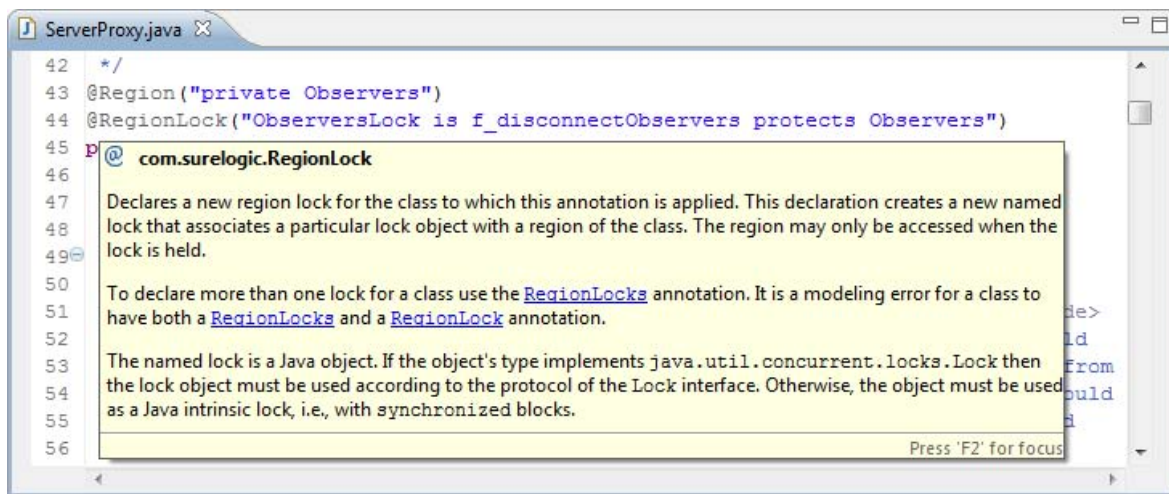
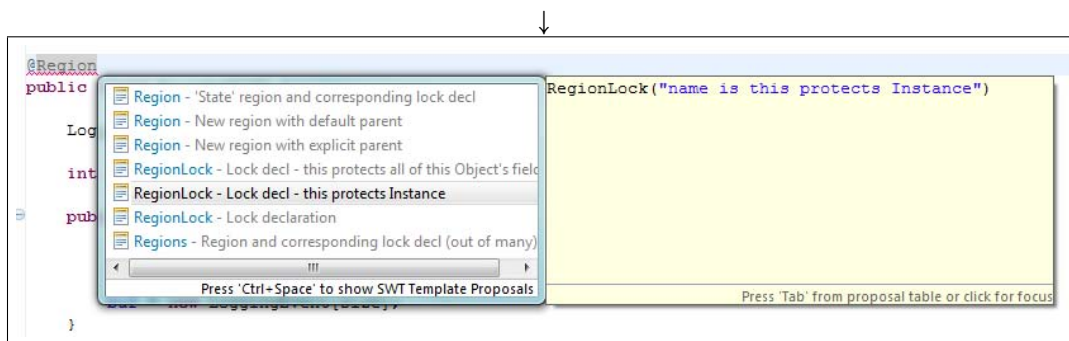


Figure 3.3: “Pop up” Javadoc about the `@RegionLock` promise in the Eclipse Java editor. The Javadoc helps the tool user understand how to use the promise correctly.

The programmer enters a portion of the promise name and examines the templates that “pop up”



The programmer selects a template to automatically add one or more promises to the code

```

@RegionLock("name is this protects Instance")
public class BoundedFIFO {

    LoggingEvent[] buf;

    int numElts = 0, first = 0, next = 0, size;
  
```

The programmer the fills out the template, naming the lock policy “FIFOLock”

```

@RegionLock("FIFOLock is this protects Instance")
public class BoundedFIFO {

    LoggingEvent[] buf;

    int numElts = 0, first = 0, next = 0, size;
  
```

Figure 3.4: Using a template to help with the syntax of the `@RegionLock` promise.

annotations to the code by selecting one or more of them in this view and selecting **Add promises to code...** from the context menu. (This interaction was illustrated in Figure 1.16 on page 26.)

Any annotation that is not well-formed, *i.e.*, it contains a syntax or semantic error, is listed in the *Modeling Problems* view. The tool user uses the information in this view to fix the problem and make the annotation well-formed.

The tool output, as a key component of the user interface of JSure, helps to support all of our principles related to practicability, especially the *incremental reward principle* (Section 1.6.1).

The verifying analyses, similar to the Eclipse Java compiler, are incremental and run in the background while the programmer continues to work. Thus, JSure unobtrusively monitors model-code consistency as a programmer works, and it provides quick feedback as a programmer works to express models.

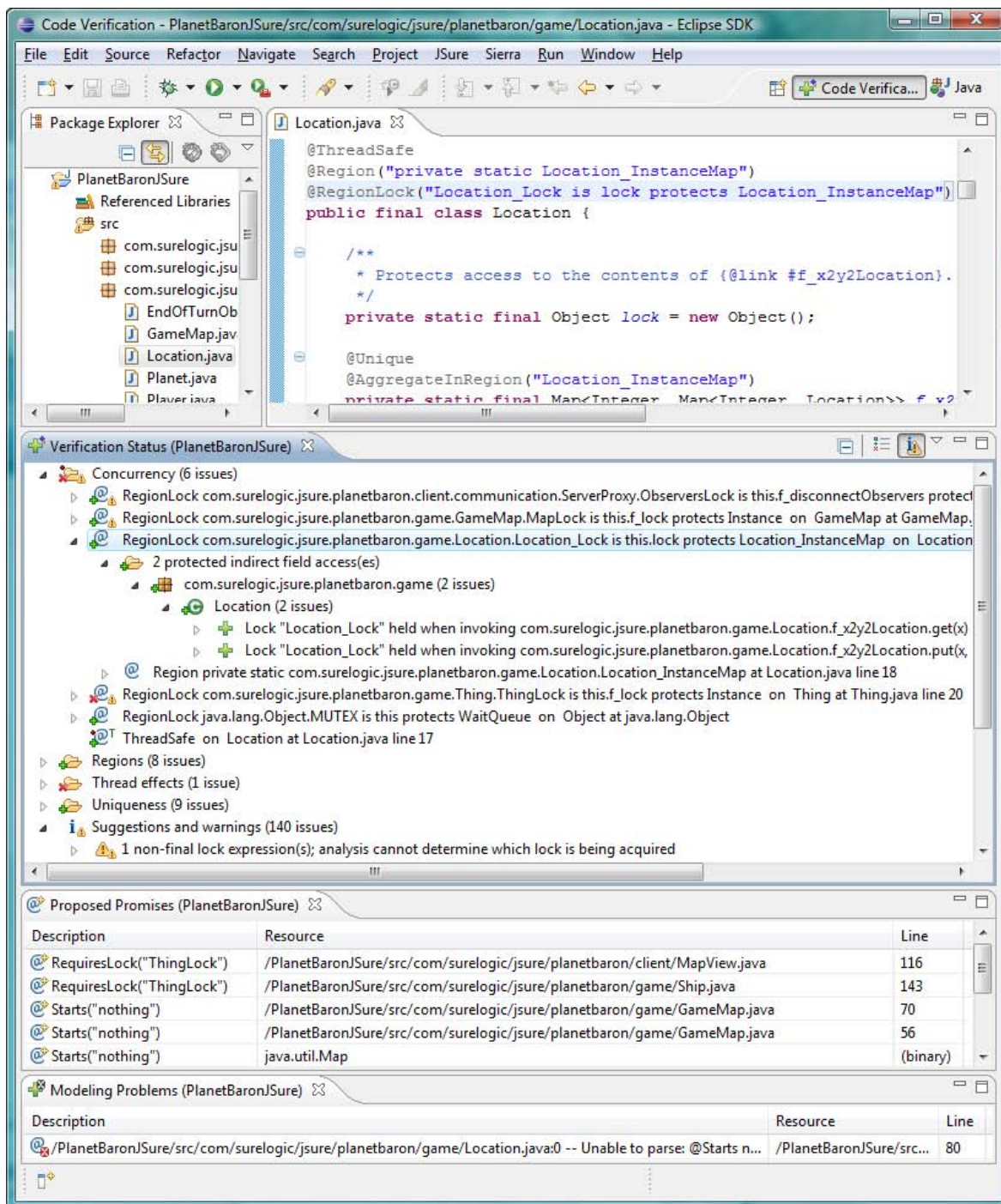




















Figure 3.5: The Code Verification perspective of the JSure analysis-based verification tool within the Eclipse Java IDE. (Top) The tool user enters annotations, e.g., `@Region` or `@Unique`, into their Java code to express a model of design intent about the program’s implementation. (Middle) After saving the annotated code, the tool reports model–code consistency in the *Verification Status* view. A “+” indicates a conservative finding of model–code consistency and an “x” indicates model–code inconsistency (or a false positive). (Bottom) Promises being proposed to the tool user are displayed in the *Proposed Promises* view. Any annotations that are not well-formed are reported to the user in the *Modeling Problems* view.


Icons used in the *Verification Status* view

Image	Description
	Category of verification results
	Promise about the code
	Consistent analysis result
	Inconsistent (or false positive) analysis result
	 analysis result vouched for with @Vouch
	An analysis result with a choice of possible prerequisite assertions
	One particular choice for a prerequisite assertion
	Informational message – a possible next modeling step
	Warning message – a possible next modeling step

Icon decorators used in the *Verification Status* view

Image	Position	Description
	lower-left	Proved consistent verification decorator
	lower-left	Not proved consistent verification decorator
	upper-left	Red dot decorator – a @Vouch , @Assume , or trusted promise is used in the verification proof
	upper-right	Trusted promise decorator – not checked by analysis
	upper-right	Virtual promise decorator – created by @Promise
	upper-right	Assume promise decorator – created by @Assume
	lower-right	Information decorator – highlights an informational message
	lower-right	Warning decorator – highlights a warning message

Icons used in the *Proposed Promises* view

Image	Description
	A promise that has been proposed by the tool to be annotated into the code

Icons used in the *Modeling Problems* view


Image	Description
	Modeling problem – check the syntax of your annotations

Figure 3.6: A guide to the iconography used by the JSure tool. (Top) The icons used in the tree presented in the *Verification Status* view. (Middle-top) The smaller images that decorate the above icons in the tree presented in the *Verification Status* view. (Middle-bottom) The icon used to indicate a proposed promise in the *Proposed Promises* view. (Bottom) The icon used to indicate modeling problems discovered by promise “scrubbing” in the *Modeling Problems* view.

3.3 Tool architecture

A component and connector shared-data view of the JSure analysis-based verification system is shown in Figure 3.7. We use this view to highlight several aspects of our design.

3.3.1 Java code representation

The *IDE* component in Figure 3.7 represents the Eclipse Java IDE. An *IDE-specific adapter* is used to interact with Eclipse through its *IDE project interface*. A project in Eclipse groups together a set of Java compilation units and libraries under a programmer-given name. The IDE project interface allows us to monitor the state of the code being worked on within each open project by the programmer. It also gives JSure access to the Eclipse representation used for Java code. We refer to this representation as the eAST, the Eclipse-based Java Abstract Syntax Tree. The eAST is an AST produced by Eclipse and used by the Java editor for code formatting, context assist, and refactoring—it is not used for compilation. This tree representation of Java code “leans” toward a concrete syntax and can parse all, or portions of, many illegal Java compilation units, *e.g.*, programs that will not type check or that contain unparsable declarations. This design choice allows the Eclipse Java editor to operate on very “rough” Java code as the programmer types. The trade-off made for this robustness is that the eAST is not well suited to semantic program analysis. All uses in the eAST are simply represented as `SimpleName` or (worse) `Expression` and binding is required to differentiate, for example, types and packages from fields and local variables¹.

The trade-offs made in its design make the eAST a poor structure for JSure program analyses to use. Therefore, JSure represents each Java compilation unit as a fAST, the Fluid IR-based Java Abstract Syntax Tree. The fAST is an AST represented in the Fluid IR that is purposely designed for semantic program analysis—including those that are flow-sensitive. This tree is very abstract to simplify analysis and will not accept illegal Java compilation units. The *promise-aware incremental parser/binder* shown in Figure 3.7 exposes an interface to the Eclipse adapter that allows parsing and binding of a Java compilation unit, either from the source text or an eAST, to create a *forest of ASTs annotated with promises*. The “forest” is composed of a fAST for each Java compilation unit in the Eclipse project being examined.

3.3.2 Promise “scrubbing”

The shared-data style of JSure is referred to as a *blackboard* [89, 88, 99] because the shared-data store shown in Figure 3.7 informs data consumers of the arrival of interesting data. Changing the *forest of ASTs annotated with promises* triggers several actions. The first involves the *promise management* component checking that each promise in the forest of ASTs is well-formed—a process we refer to as promise scrubbing.

If a promise has a syntactic or semantic problems then it is not considered well-formed. For example, consider the simple “variable” class, `Var`, annotated with object-oriented effects [54] promises in Figure 3.8. Several of the promises in this class are not well-formed. The `@RegionEffects("read Value")` promise at line 11 has a syntax problem. The syntax of the `@RegionEffects` requires the token `reads` rather than `read`. The `@InRegion("Valuee")`

¹Eclipse uses the name Java DOM/AST rather than eAST for the set of classes that it uses to model the source code of a Java program as a structured document (Google: `org.eclipse.jdt.core.dom`).

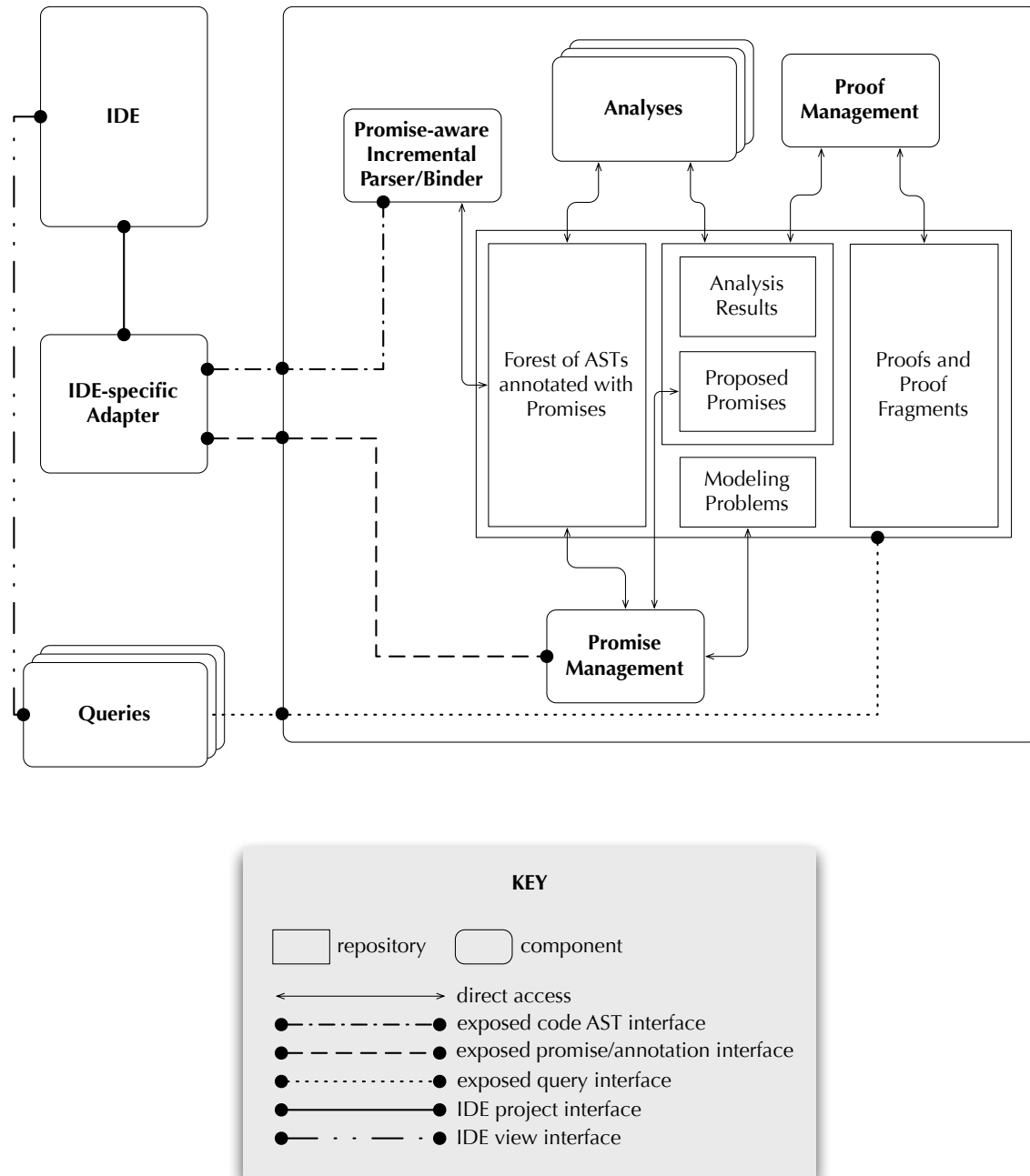


Figure 3.7: Component and connector shared-data view of the JSure analysis-based verification system.


```

1 @Region("public Value")
2 public class Var {
3   @InRegion("Valuee")           // Broken - unknown region "Valuee"
4   private int value;
5
6   @RegionEffects("none")
7   public Var(int v) {
8     value = v;
9   }
10
11  @RegionEffects("read Value") // Broken- syntax error, expected "reads"
12  public int getValue() {
13    return value;
14  }
15
16  @RegionEffects("writes Valu") // Broken - unknown region "Valu"
17  public void setValue(int v) {
18    value = v;
19  }
20 }

```

Figure 3.8: A class containing modeling problems that are caught during promise scrubbing.

promise at line 3 and the `@RegionEffects("writes Valu")` promise at line 16 are not well-formed because the regions they refer to are unknown—the region declared at line 1 was named `Value`. These two promises are syntactically correct but cannot be given meaning in the context that they appear.

Promise scrubbing is a specialized program analysis that is performed by the *promise management* component shown in Figure 3.7. The analysis examines the forest of ASTs and records any modeling problems found into the *modeling problems* repository. If a promise is well-formed then the promise management component constructs a promise drop to represent it. The promise drop is then linked to the declaration within the fAST where the promise was annotated. A promise drop is an identity in drop-sea for a well-formed promise, analogous to the promise symbols used in the previous chapter.

The various drops and their role in drop-sea are further elaborated in Section 3.4. In this section we sketch their use as we overview the tool architecture.

3.3.3 Analysis of the “forest”

When the *forest of ASTs annotated with promises* in Figure 3.7 contains one or more “scrubbed” well-formed promises, each identified by a promise drop, this triggers analysis of the forest by constituent verifying analyses (e.g., lock policy, uniqueness, effects upper bounds, thread effects). We do not require every promise in the forest to be well-formed—bad promises are ignored by the constituent analyses. As described in the previous chapter, verifying analyses produce two outputs: *analysis results* and *proposed promises*.

Analysis results are reported from a verifying analysis via result drops. When a constituent analysis makes a judgment about the consistency of a particular promise it constructs a result drop about that promise. The analysis sets, on the drop, whether the result represents a conservative finding of consistency or not. It provides a reference to what portion

or place in the code (by providing a reference to a fAST node) the result is about. Finally, the analysis provides the prerequisite assertion for the result in terms of promise drops or, as is encouraged in Chapter 2, proposed promise drops. Proposed promises are represented via a proposed promise drop which represents the syntax of the desired promise as well as a reference to a declaration within the fAST where the promise should be located.

Analysis of the forest of ASTs is performed by the *Analyses* component shown in Figure 3.7.

3.3.4 Promise matching

The tool engineering of JSure supports promise matching, however, as of this writing most analysis still perform promise matching themselves because the analyses predate the existence of this generic functionality. The analysis determines the promise that it requires as part of its prerequisite assertion and checks if it exists. If so, then the promise drop is passed to the result drop. If not, then the analysis constructs a proposed promise drop and passes it to the result drop. This is one area where the tool engineering is “catching up” to the theory presented in Chapter 2. However, from the tool user’s point of view the result is the same—wherever possible proposed promises are “matched” with real promises and the set of remaining proposed promises is used to allow the JSure user interface to prompt the user to consider adding “missing” promises to the code.

Promise matching is performed by the *promise management* component shown in Figure 3.7.

3.3.5 Promise verification

Promise drops and analysis-reported result drops yield a graph stored in the *proofs and proof fragments* repository shown in Figure 3.7. This graph is analyzed by the *proof management* component to determine, based upon the reported analysis results, the consistency of each promise. The algorithm used to do this is an automation of the verification proof calculus presented in the previous chapter.

The objective, however, is not just to determine the consistency of each promise, but also to be able to “explain” to the tool user why a promise is verifiable or why it is unverifiable. Therefore, the graph represents the structure of all the possible verification proofs as well as connections to all the supporting analysis results.

3.3.6 Querying the tool results

The *queries* component shown in Figure 3.7 populates the Eclipse views with the information contained in the shared repositories. A mapping from the repositories (shown in Figure 3.7) to their corresponding Eclipse views (shown in Figure 3.5) is illustrated in Figure 3.9.

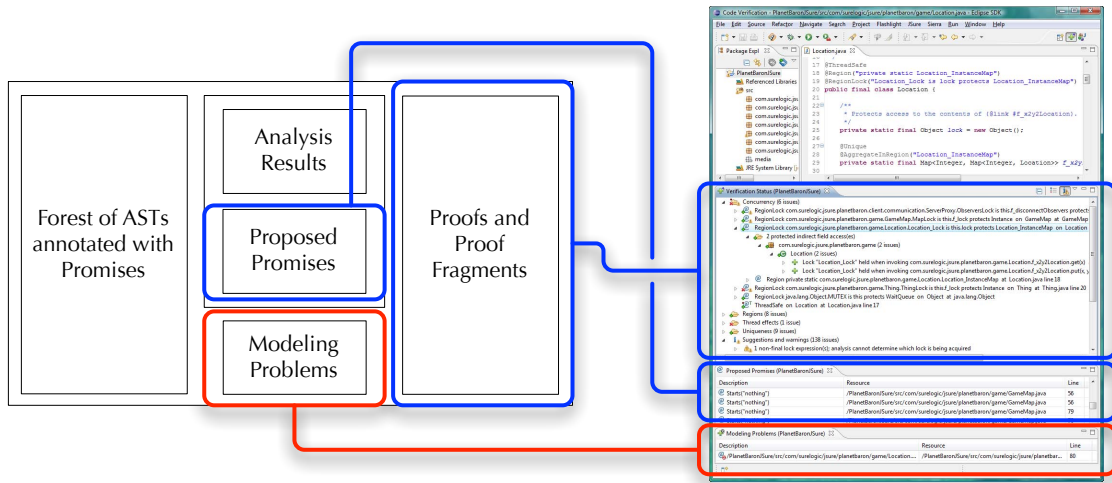


Figure 3.9: Mapping from the repositories shown in Figure 3.7 to the Eclipse views for the JSure tool.

3.4 The drop-sea proof management system

This section introduces *drop-sea*, the proof management system used by the JSure tool. Drop-sea is our name for the *proof management* component shown in Figure 3.7, however, we often refer to the shared repositories that this component interacts with by the name drop-sea as well. For example, we may state that a result drop is reported into drop-sea when we mean, more precisely by the view of the system shown in Figure 3.7, that it is placed in the *analysis results* shared repository thereby triggering action by the *proof management* component on this result. This imprecise use of the term drop-sea occurs for the *analysis results*, *proposed promises*, and *proofs and proof fragments* repositories.

We start this section by describing how analysis results are represented as a graph in drop-sea. In particular, how the tabular analysis results shown in Chapter 2 relate to their representation in drop-sea, this includes showing how drop-sea represents results about recursive code and results that reference “unmatched” proposed promises as part of their prerequisite assertion. We continue by discussing how the three types of unverified contingencies are represented in drop-sea: (1) programmer vouches, (2) disabled analyses, and (3) local assumptions. With this background, we are able to present our algorithm for the automation of the verification proof calculus presented in the previous chapter. We end the section by presenting how drop-sea performs the role of truth maintenance for the system by tracking dependencies between code, promises, analysis results, and other types of “drops” introduced by the various components that make up the tool.

3.4.1 Representing analysis results

Our representation of analysis results and the promises that they verify is motivated by the observation that an understanding about how particular promises relate to one another is embodied in the constituent analyses. This fact allows a clear separation of concerns between drop-sea and the constituent analyses. The constituent analyses are concerned with

the semantics of the promises that they verify as well as producing sound analysis results based upon the semantics of the Java programming language. Drop-sea is concerned with automated reasoning to produce a sound verification result for each promise and with tracking dependencies to keep these results up to date as the code and promises about the code change. This separation is key to our ability to “scale up” in terms of adding new promises and verifying analyses.

In our approach, constituent analyses report their results to drop-sea. These results are modeled as a graph. We return to the running example used in the previous chapter from the `util.concurrent` library. The `util.concurrent` code is shown in Figure 2.4 on page 54. Figure 3.10 shows the drop-sea graph for the `SynchronizedVariable` and `SynchronizedBoolean` classes after the lock policy, effects, thread effects, and uniqueness analyses have reported their analysis results. The structure shown in Figure 3.10 is a tree, however, in the presence of recursive calls in the code the resulting structure is a graph (a recursive example is shown below).

Figure 3.11 shows the tabular analysis results used to construct the drop-sea graph in Figure 3.10. Each row in Figure 3.11 becomes an analysis result drop in the graph (represented as a rectangle). The *Finding* column in the tabular representation is shown in the drop-sea graph as either a “+” for a conservative judgment by the analysis of consistency, or an “x” otherwise. A directed edge from a result drop to a promise drop indicates the promise that the result is about and matches the *About* column in the tabular representation.

The representation in the drop-sea graph of the promise logic formula in the *Prerequisite* column in the table is more complex. A directed edge from a promise drop to a result drop indicates that promise is part of the prerequisite assertion formula for that result. For example, the prerequisite assertion for f_1 is shown in the table as the formula r_{29} . A directed edge from r_{29} to f_1 is used to represent this formula in the drop-sea graph (at the bottom-center of Figure 3.10). Multiple edges directed to a result indicate a conjunction, *i.e.*, \wedge . Two special nodes are used to represent disjunction, *i.e.*, \vee . These are the *choice* and *choice option* nodes that are represented in Figure 3.10 as a diamond and a rounded rectangle, respectively. For example, the prerequisite assertion for f_8 is shown in the table as the promise logic formula $r_{20} \vee (r_{18} \wedge r_{19})$. This formula is represented in Figure 3.10 as one choice node, c_1 , and two choice option nodes, o_1 and o_2 . Drop-sea, therefore, represents prerequisite assertions in disjunctive normal form.

If no directed edge exists in the drop-sea graph from a promise drop to a particular result drop (or from a choice drop to a particular result drop) this indicates that the prerequisite assertion formula in the tabular analysis results for that result is \top —indicating no constraint on that result.

3.4.2 Representing recursion

Drop-sea supports the verification of promises about recursive code. Figure 3.12 uses the `Fibonacci` class from Chapter 2 to illustrate what a drop-sea graph looks like in the presence of recursive code. The graph is constructed from the table of analysis results in the manner described above, however, in the case of the `Fibonacci` class it contains cycles. This is because the recursive calls made within the `fib` method cause two of the analysis results to use the same promise as both their prerequisite assertion and their consequential assertion.

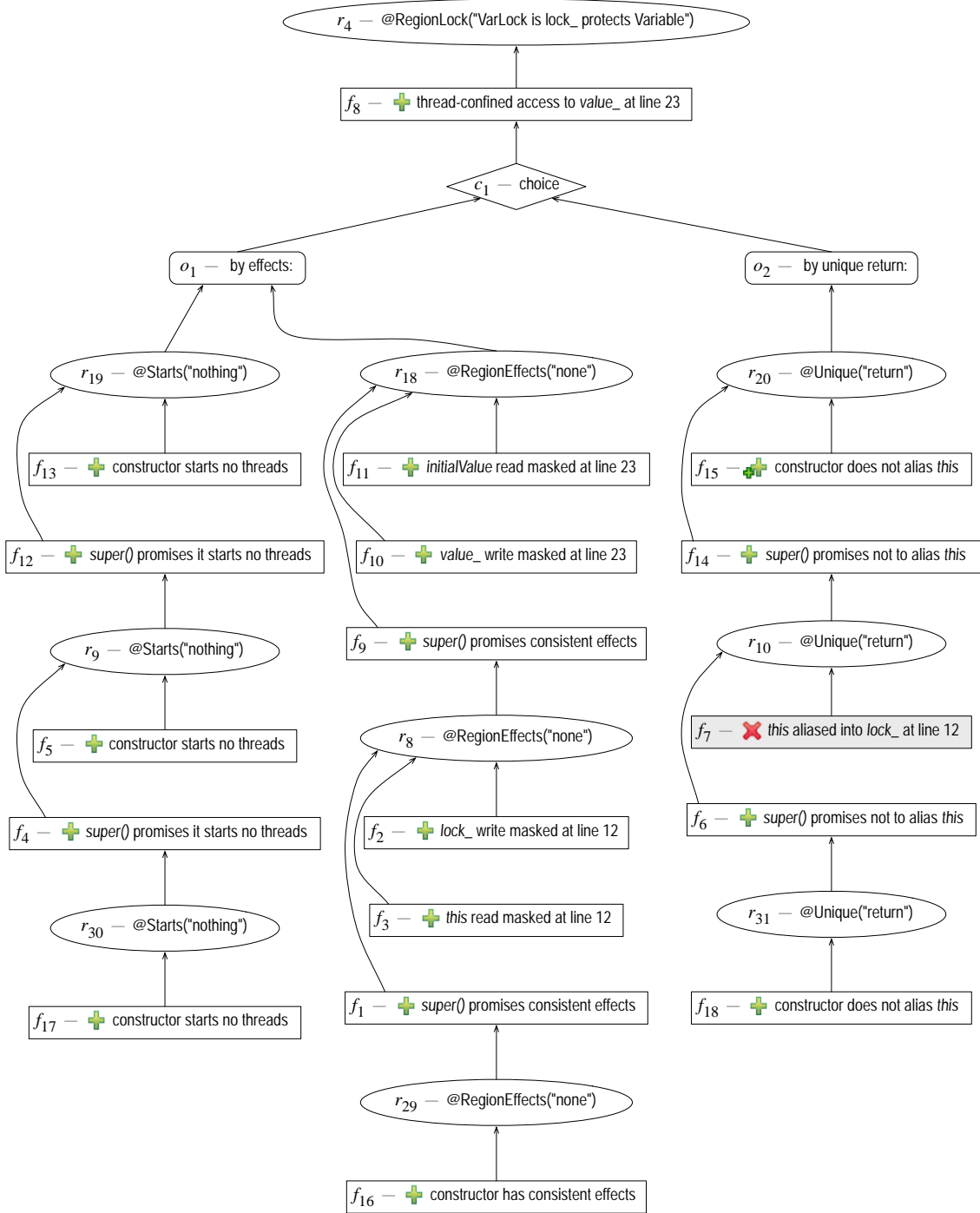


Figure 3.10: Drop-sea graph for the `SynchronizedVariable` and `SynchronizedBoolean` classes after the lock policy, effects, thread effects, and uniqueness analyses have reported analysis results for the promises and code in Figure 2.4. Promise drops, which are independent of any specific analysis, are represented by ovals. Result drops are represented as rectangles. A “+” indicates a consistent analysis result. An “x” indicates an inconsistent or false positive analysis result (boxed in grey). The choice and choice options are represented by a diamond and a rounded rectangle, respectively. A directed edge from a promise drop to a result drop indicates that the promise is a prerequisite assertion for that result. A directed edge from a result drop to a promise drop indicates that the result is about that promise.

Analysis Results				
	Finding	About	Prerequisite	Description
f_1	+	r_8	r_{29}	<code>super()</code> promises consistent effects
f_2	+	r_8	\top	<code>lock_</code> write masked at line 12
f_3	+	r_8	\top	<code>this</code> read masked at line 12
f_4	+	r_9	r_{30}	<code>super()</code> promises it starts no threads
f_5	+	r_9	\top	constructor starts no threads
f_6	+	r_{10}	r_{31}	<code>super()</code> promises not to alias <code>this</code>
f_7	×	r_{10}	\top	<code>this</code> aliased into <code>lock_</code> at line 12
f_8	+	r_4	$r_{20} \vee (r_{18} \wedge r_{19})$	thread-confined access to <code>value_</code> at line 23
f_9	+	r_{18}	r_8	<code>super()</code> promises consistent effects
f_{10}	+	r_{18}	\top	<code>value_</code> write masked at line 23
f_{11}	+	r_{18}	\top	<code>initialValue</code> read masked at line 23
f_{12}	+	r_{19}	r_8	<code>super()</code> promises it starts no threads
f_{13}	+	r_{19}	\top	constructor starts no threads
f_{14}	+	r_{20}	r_{10}	<code>super()</code> promises not to alias <code>this</code>
f_{15}	+	r_{20}	\top	constructor does not alias <code>this</code>
f_{16}	+	r_{29}	\top	constructor has consistent effects
f_{17}	+	r_{30}	\top	constructor starts no threads
f_{18}	+	r_{31}	\top	constructor does not alias <code>this</code>

Figure 3.11: Analysis results for the `util.concurrent` code in Figure 2.4 after the results of promise matching have been used to replace proposed promises that appear in each prerequisite assertion with “matched” real promises.

3.4.3 Representing unmatched proposed promises

To this point we have avoided a discussion of what is done when an analysis proposes a promise that does not exist as a real promise. In this situation there are two reasonable design choices:

1. Mark the result as an “×”. Present the tool user with the “missing” promises for addition into the code base to eliminate the “×”.
2. Show the analysis result as a “+” but mark it as contingent, *i.e.*, with a red dot. Present the tool user with the “missing” promises for addition into the code base to eliminate the red dot.

The current JSure implementation implements the first design option. An example of this design choice, from the tool user’s point of view, is presented in Figure 1.16 on page 26 as part of our description of the tool interaction when using proposed promises to automatically annotate the `BoundedFIFO` class from a single `@RegionLock` promise. When the `@RegionLock` promise is the only promise in the `BoundedFIFO` code the analysis results (and therefore the consistency of the promise) were all “×”s—not “+”s with a red dot.

We suggest that this design choice is more effective in encouraging tool users to complete partially annotated models. Programmers appear to be more motivated to change an “×” to a “+” than they are to eliminate the red dot on a “+”. That said, there is no reason that

```

1 public class Fibonacci {
2   long callsToFibMethod = 0;
3
4   @Borrowed("this")
5   public long fib(int n) {
6     callsToFibMethod++;
7     if (n <= 1)
8       return n;
9     else
10      return fib(n-1) + fib(n-2);
11   }
12 }

```

Analysis Results for Fibonacci

	Finding	About	Prerequisite	Description
f_1	+	r_4	r_4	<code>fib(n-1)</code> at line 10 promises not to alias <code>this</code>
f_2	+	r_4	r_4	<code>fib(n-2)</code> at line 10 promises not to alias <code>this</code>
f_3	+	r_4	\top	<code>fib(int)</code> does not alias <code>this</code>

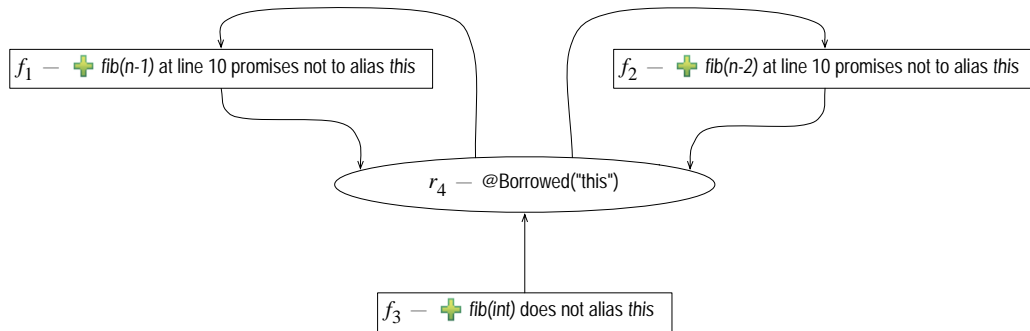


Figure 3.12: An example of cycles in a drop-sea graph due to recursive calls in the program being verified. (Top) The Java code for (inefficiently) computing a Fibonacci number using recursive calls. The `Fibonacci` instance tracks the number of calls to `fib`. The `fib` method promises that it does not alias the receiver. (Middle) Analysis results after the results of promise matching have been used to replace proposed promises that appear in each prerequisite assertion with “matched” real promises. (Bottom) The drop-sea graph for the `Fibonacci` class.

the tool, in the future, can't implement both approaches and allow the user to elect the one that they prefer. The second option, however, drifts into verifying based on annotations that aren't there. This moves away from verification based on explicitly expressed design intent.

Figure 3.13 uses the `TravelAgentBean` class from Chapter 2 to illustrate a drop-sea graph with unmatched proposed promises. These drops allow the tool to represent the “weakest” prerequisite assertion for a particular promise and to propose annotations to the code base that can be reviewed and accepted by the tool user. A tool view showing the presentation of the three proposed promises about `TravelAgentBean` to the user is shown in Figure 3.14.

3.4.4 Representing contingencies

Figure 3.15 shows a short snippet of code written by Goetz to illustrate a use of `volatile` fields referred to as the *cheap read-write lock trick*². This “trick” is so named because you are using different synchronization mechanisms for reads and writes. Declaring a field to be `volatile` ensures the visibility of the current value when reading, however, a lock is used to ensure that the increments to the counter are atomic. If reads are more common than increments you may be able to get a higher degree of sharing than if you used locking for both operations.

As of this writing, the JSure tool is not able to specify and verify a locking policy implemented in this manner. (JSure could verify the locking policy of this class if it used a `ReadWriteLock` rather than the cheap read-write lock trick—a less fragile approach that we would recommend.) We can, however, specify how locking is intended to be done and vouch for reads of the `volatile` field.

We specify the locking policy for `Counter` using the `@GuardedBy` promise at line 8. This promise was proposed by Goetz, *et al.* in [51] and can be verified by the JSure tool³. This specification is, however, imperfect because the read of `value` at line 11 is reported by the verifying analysis to be inconsistent with the locking model because—the lock is not held. We use the `@Vouch` annotation at line 9 to tell the tool that this inconsistency is intentional. In the drop-sea graph the vouched-for “x” analysis result is represented as a (hollow grey) “+” (rather than an “x”) with the `@Vouch` as its prerequisite assertion.

The `@ThreadSafe` promise at line 1, another promise proposed by Goetz, *et al.* in [51], is not able to be verified by the JSure tool. This promise is primarily for documentation purposes and it is “trusted” by the tool. A trusted promise is represented in drop-sea as a promise that is not supported by any analysis results. Note that, by this definition, `@Vouch` is also a trusted promise. It is used as the prerequisite assertion for an analysis result, but is not, itself, supported by any analysis results. (Trusted promises are discussed further in Section 3.6.)

3.4.5 Computing verification results

In this section we present our algorithm for the automation of the verification proof calculus presented in the previous chapter. The algorithm operates on the drop-sea graphs that we

²<http://www.ibm.com/developerworks/java/library/j-jtp06197.html>

³The equivalent Greenhouse promise is `@RegionLock("CounterLock is this protects value")`, however, because this is Goetz's code we elect to use his annotations.


```

1 @Stateless
2 public class TravelAgentBean implements TravelAgentRemote {
3
4     @PersistenceContext(unitName="titan") private EntityManager manager;
5
6     @Starts("nothing")
7     public void createCabin(Cabin cabin) {
8         if (findCabin(cabin.getId()) == null)
9             manager.persist(cabin);
10    }
11
12    public Cabin findCabin(int pKey) {
13        return manager.find(Cabin.class, pKey);
14    }
15 }

```

Analysis Results for TravelAgentBean

	Finding	About	Prerequisite	Description
f_1	+	r_6	\top	<code>createCabin</code> starts no threads
f_2	+	r_6	q_1	<code>findCabin</code> promises it starts no threads
f_3	+	r_6	q_2	<code>getId</code> promises it starts no threads
f_4	+	r_6	q_3	<code>persist</code> promises it starts no threads

Proposed Promises		
	Promise	On
q_1	<code>@Starts("nothing")</code>	<code>findCabin(int)</code> at line 13
q_2	<code>@Starts("nothing")</code>	<code>getId()</code> in class <code>Cabin</code>
q_3	<code>@Starts("nothing")</code>	<code>persist(Object)</code> in class <code>EntityManager</code>

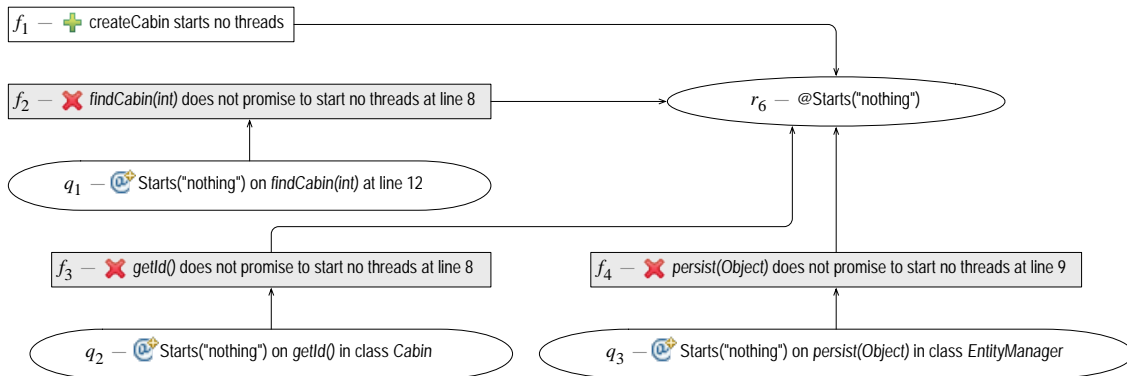


Figure 3.13: An example of unmatched proposed promises in a drop-sea graph due to missing design intent in the code base being verified. (Top) The Java code for an EJB 3.0 stateless session bean, `TravelAgentBean`, that promises that the `createCabin` method will not start any threads. (Middle-top) Analysis results for the `TravelAgentBean` class. (Middle-bottom) Proposed promises for the `TravelAgentBean` class. (Bottom) The drop-sea graph for the `TravelAgentBean` class showing (1) proposed promise drops, e.g., q_1 , q_2 , and q_3 , are created in drop-sea for each proposed promise if no “matching” real promise can be found and (2) result drops that have a proposed promise drop as part of their prerequisite assertion are (always) “x” results.

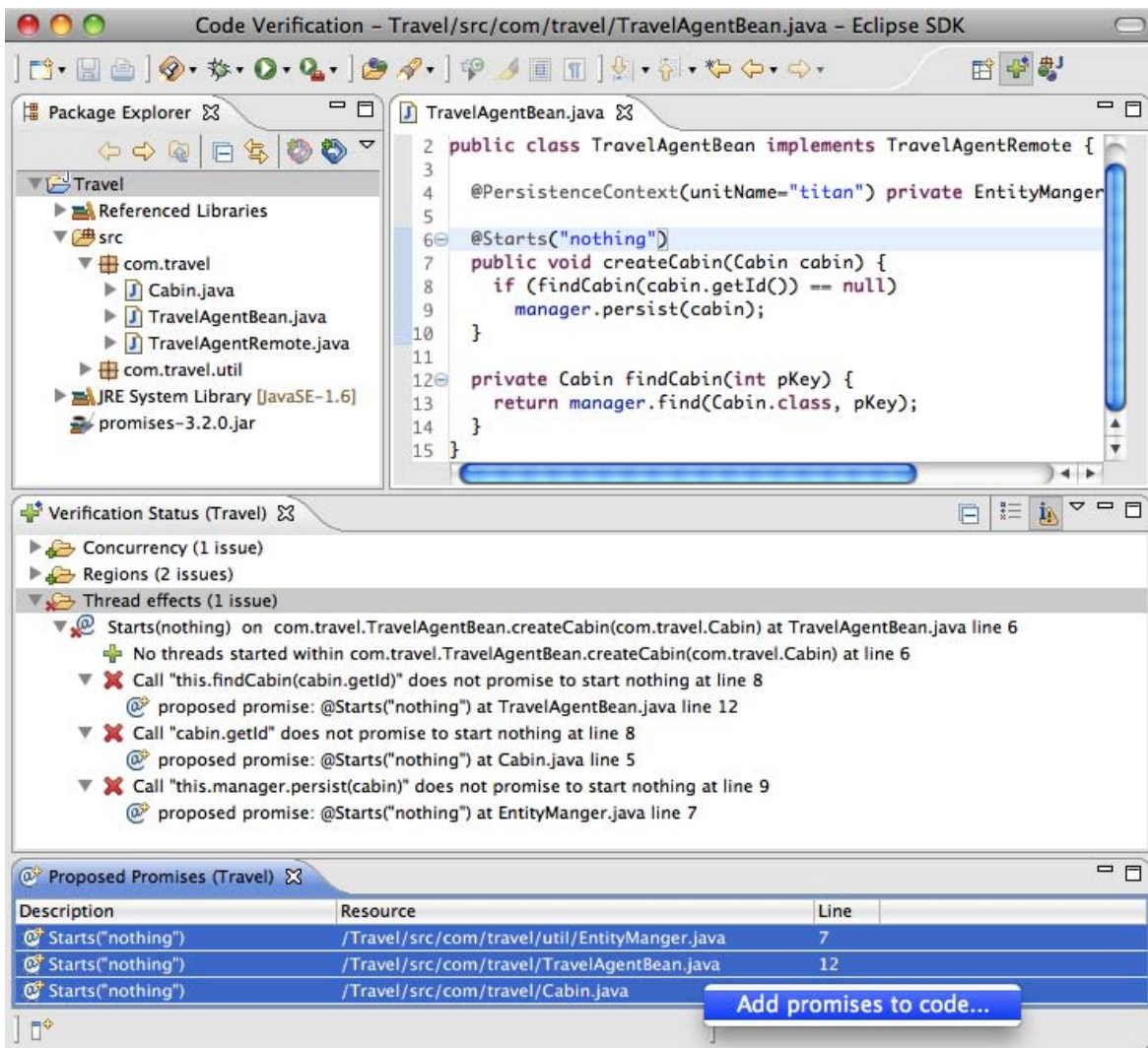


Figure 3.14: Tool view proposing three “missing” promises to help make the `@Starts("nothing")` promise at line 6 on the `createCabin` method consistent. In the *Verification Status* view each proposed promise is shown below the analysis result that reported it. The *Proposed Promises* view collects together all of the proposed promises for the tool user to examine. A context menu (shown at the bottom-right) is available in both views to allow the user to request that one or more of the proposed promises be automatically added to the code.

```

1 @ThreadSafe
2 public class Counter {
3
4     // Employs the cheap read-write lock trick
5     // All mutative operations MUST be done with the "this" lock held
6     @GuardedBy("this")
7     private volatile int value;
8
9     @Vouch("cheap read-write lock trick")
10    public int getValue() {
11        return value;
12    }
13
14    public synchronized int increment() {
15        return value++;
16    }
17 }

```

Analysis Results for Counter

	Finding	About	Prerequisite	Description
f_1	×	r_6	\top	lock not held when accessing <code>value</code> at line 11
f_2	+	r_6	\top	lock held when accessing <code>value</code> at line 15

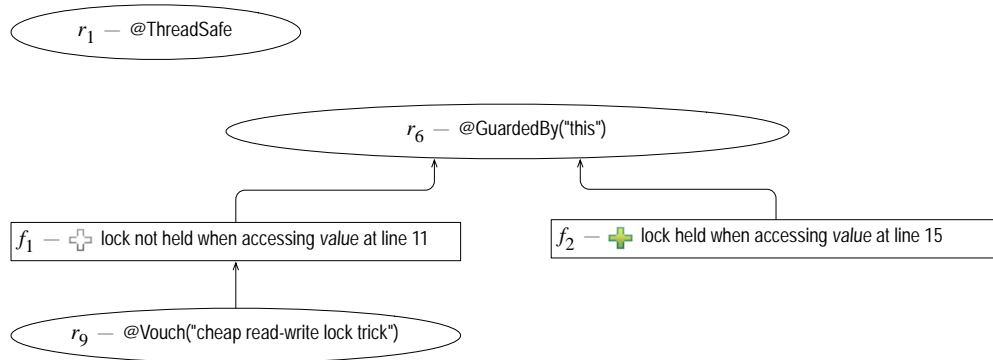


Figure 3.15: An example of (1) vouching for an “×” analysis result and (2) a trusted promise in a drop-sea graph. (Top) The Java code for a counter that employs the “cheap read-write lock trick.” (Middle) Analysis results for the `Counter` class. (Bottom) The drop-sea graph for the `Counter` class showing (1) the “×” analysis result, r_1 has been vouched for as indicated by a (hollow grey) “+” and r_9 as its prerequisite assertion and (2) the trusted `@ThreadSafe` promise which is not supported by any analysis results.

Meaning		Computed Result	UI Decoration	
Verified	w/Contingency	$V(n)$	lower-left	upper-left
Yes		$(true, true)$	+	
Yes	Yes	$(true, false)$	+	•
		$(false, true)$	×	
	Yes	$(false, false)$	×	•

Figure 3.16: Meaning of computed verification results and the icon and position for the decorators used in the tool *Verification Status* view.

presented in the previous section.

A *drop-sea graph* is a finite directed graph containing four types of nodes: model nodes, result nodes, choice nodes, and choice option nodes. Our algorithm computes a result for each node in the graph, $V(n)$ (the verification result at node n), of the form $\mathcal{P}(\{true, false\} \times \{true, false\})$, *i.e.*, a tuple of two Boolean values, where the first value indicates if the model is consistent and the second value indicates if the result is checked. The negation of the second value indicates the need for a red dot. The meaning of the four possible results are shown in Figure 3.16 alongside the user interface icons used to decorate results in the *Verification Status* view.

We specify our algorithm in the style presented by Nielson, Nielson, and Hankin in [87]. Our formalization of the drop-sea graph along with the definitions we require are given in Figure 3.17. The definitions at the top formalize the drop-sea graph, our algorithm will be most concerned with $\text{flow}^R(\mathbf{S})$, the reverse flows in the graph, because it calculates its results in this direction. Four functions are defined to identify the node type in the graph, *e.g.*, $\text{isResult}(n)$ to identify a result node, and so on. The $\text{consistent}(n)$ function indicates if a node is a consistent result node, *i.e.*, the analysis result reported was consistent. The value reported by this function does not consider if an “×” result has been vouched for via an @Vouch promise. This is indicated by the $\text{vouch}(n)$ function. The first and second functions are used to return the first or second constituent *true* or *false* value from the result for a node, *i.e.*, $V(n)$, respectively.

Figure 3.18 shows the flow equations for our computation. These equations are applied for each node in the drop-sea graph until a fixed point is reached. Our implementation uses a worklist algorithm [83], traditionally used for iterative data-flow analysis, to iterate the flow equations. We first discuss the equations presented in Figure 3.18 and then the worklist implementation used by JSure.

The flow equations given in Figure 3.18 are precise, but slightly inscrutable. Their behavior, however, is straightforward. We discuss the flow equation for each type of node below.

- $\text{isResult}(n)$: The flow equations for result drop nodes.
 - **Verified:** *True* if the analysis result is consistent, or vouched for, and all nodes with a directed edge toward this node are verified, *false* otherwise.
 - **Red dot:** *Yes* (*i.e.*, *false*) if the analysis result is vouched for or any node with a directed edge toward this node has a red dot, *no* (*i.e.*, *true*) otherwise.
- $\text{isPromise}(n)$ or $\text{isOption}(n)$: The flow equations for promise or choice option nodes.

definitions and functions

\mathbf{S}	<i>def</i>	the drop-sea graph
$\text{nodes}(\mathbf{S})$	<i>def</i>	the set of nodes in the drop-sea graph
n	\in	$\text{nodes}(\mathbf{S})$
$\text{flow}(\mathbf{S})$	<i>def</i>	The set of all edges, or <i>flows</i> , in the graph; of the form $\wp(\text{nodes}(\mathbf{S}) \times \text{nodes}(\mathbf{S}))$
$\text{flow}^R(\mathbf{S})$	$=$	$\{(n, n') \mid (n', n) \in \text{flow}(\mathbf{S})\}$
$\text{isResult}(n)$	$=$	<i>true</i> if n identifies a result node, <i>false</i> otherwise
$\text{isPromise}(n)$	$=$	<i>true</i> if n identifies a promise node, <i>false</i> otherwise
$\text{isChoice}(n)$	$=$	<i>true</i> if n identifies a choice node, <i>false</i> otherwise
$\text{isOption}(n)$	$=$	<i>true</i> if n identifies a choice option node, <i>false</i> otherwise
$\text{consistent}(n)$	$=$	$\begin{cases} \text{true} & \text{isResult}(n) \wedge n \text{ represents a consistent analysis result} \\ \text{false} & \text{otherwise} \end{cases}$
$\text{vouch}(n)$	$=$	$\begin{cases} \text{true} & \text{isResult}(n) \wedge n \text{ is vouched for by the programmer} \\ \text{false} & \text{otherwise} \end{cases}$
$\text{first}(x, y)$	$=$	x
$\text{second}(x, y)$	$=$	y

Figure 3.17: Definitions and functions for our verification analysis.

- **Verified:** *True* if all nodes with a directed edge toward this node are verified, *false* otherwise.
- **Red dot:** *Yes* (*i.e.*, *false*) if any node with a directed edge toward this node has a red dot *or* if there are no nodes with a directed edge toward this node, *no* (*i.e.*, *true*) otherwise.

The above predicates handles trusted promises. If there are no directed edges toward a promise node this indicates that it has no analysis results supporting its consistency. In our algorithm it is vacuously verified, *i.e.*, trusted, but this trust (a contingency) is indicated via a red dot. (For an example see Section 3.6.)

- $\text{isChoice}(n)$: The flow equations for choice nodes.

In this case we take the most desirable option presented by the choice option nodes with a directed edge toward this node. A verified choice is chosen over a choice that cannot be verified and a choice without a red dot is chosen over a choice with a red dot.

The worklist algorithm used to iterate the flow equations above continues to process nodes until it becomes empty—indicating that a fixed point has been reached. The algorithm is started with all the nodes in the drop-sea graph on the worklist. Each node on the worklist is processed. If the node being processed changes then it places all the nodes with a directed edge from the node being processed to them back on the worklist.

flow equations: $V^=$

if `isResult(n)`

$$V(n) = ((\text{consistent}(n) \vee \text{vouch}(n)) \wedge \forall x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \text{first}(x), \\ (\text{consistent}(n) \vee \neg \text{vouch}(n)) \wedge \forall x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \text{second}(x))$$

if `isPromise(n)` or `isOption(n)`

$$V(n) = (\forall x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \text{first}(x), \\ |\{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\}| > 0 \wedge \forall x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \text{second}(x))$$

if `isChoice(n)`

$$V(n) = \left\{ \begin{array}{ll} (true, true) & \text{if } \exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \\ & \text{first}(x) \wedge \text{second}(x) \\ (true, false) & \text{if } (\exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \text{first}(x)) \\ & \wedge \neg(\exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \\ & \text{first}(x) \wedge \text{second}(x)) \\ (false, true) & \text{if } (\exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \\ & \neg \text{first}(x) \wedge \text{second}(x)) \\ & \wedge \neg(\exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \\ & \text{first}(x)) \\ & \wedge \neg(\exists x \in \{V(n') \mid (n', n) \in \text{flow}^R(\mathbf{S})\} : \\ & \text{first}(x) \wedge \text{second}(x)) \\ (false, false) & \text{otherwise} \end{array} \right.$$

Figure 3.18: Flow equations for our verification analysis.

The computational efficiency of this algorithm depends upon the flow problem we are solving and the management of the worklist itself. We admit that the current implementation in `JSure` could be improved, however, we have not identified any significant performance problems with it (using several commercial Java profilers). The thread coloring analysis, developed by Sutherland [103], creates the largest drop-sea graphs that we have encountered to date. These graphs can contain tens of thousands of nodes. Sutherland does note, however, that our algorithm is slower than the underlying constituent analyses used in his approach.

We now consider the verification results computed on two of the examples discussed above: `util.concurrent` and `Counter`

Example: `util.concurrent`

The *Verification Status* view in the `JSure` user interface shows computed verification results in a tree form. A portion of the tool view of the computed verification results for the `VarLock` locking model is shown in Figure 3.19. This portion of the tool view shows the choice node in the drop-sea graph. The computed verification results for the `SynchronizedVariable` and `SynchronizedBoolean` classes is shown in Figure 3.20.

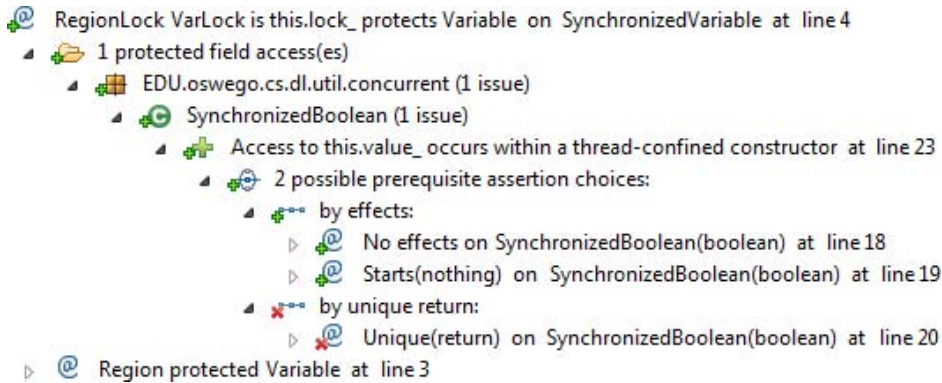


Figure 3.19: A JSure tool view showing a portion of the global verification results for the `VarLock` locking model. The view shows the portion of the drop-sea graph decorated with computed verification results to just below the choice option nodes in Figure 3.20.

The tool view of the (not verified) “by unique return” choice option is shown in Figure 3.21. The tool view of the (verified) “by effects” choice option is shown in Figure 3.22.

Example: Counter

The `util.concurrent` example above does not contain any contingencies. We now consider an example that does, the `Counter` class shown in Figure 3.15. Figure 3.23 shows the computed verification results for this class in the drop-sea graph and the tool user interface. The results for this class are based upon two trusted promises: `@ThreadSafe` and `@Vouch`. In addition, the `@GuardedBy` promise is marked with a red dot because one of the two analysis results reported by the lock analysis about the `Counter` class is an “x” result that is vouched for by the programmer.

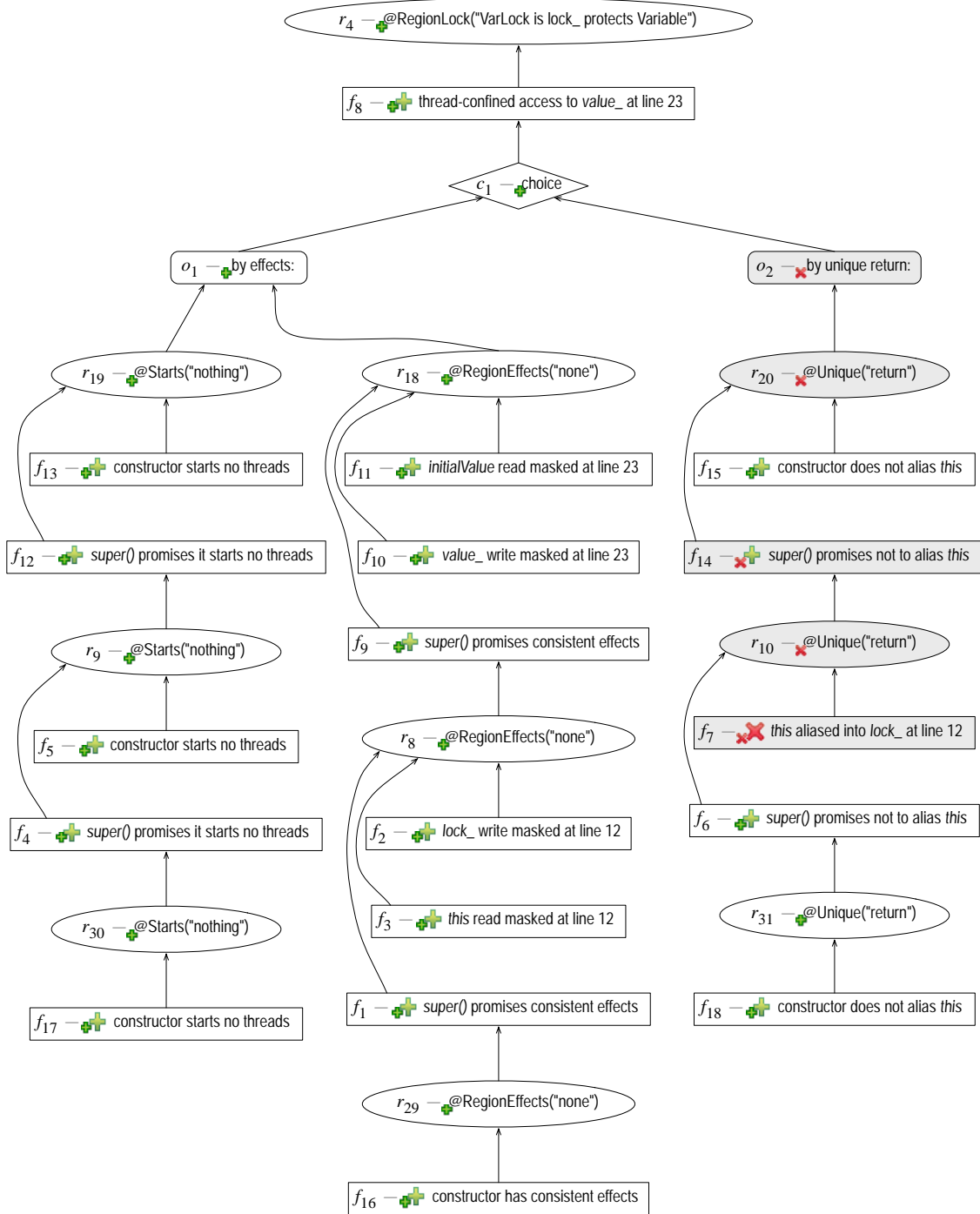


Figure 3.20: Drop-sea graph for the `SynchronizedVariable` and `SynchronizedBoolean` classes showing computed verification results. A small “+” (to the lower-left) indicates model-code consistency. A small “x” (to the lower-left) indicates a failure to prove model-code consistency (the grey nodes). The verification results are only meaningful on promise drops (which represent assertions), however, we decorate the other nodes to help the tool user track down what particular “x” result are causing a promise to be unverifiable.

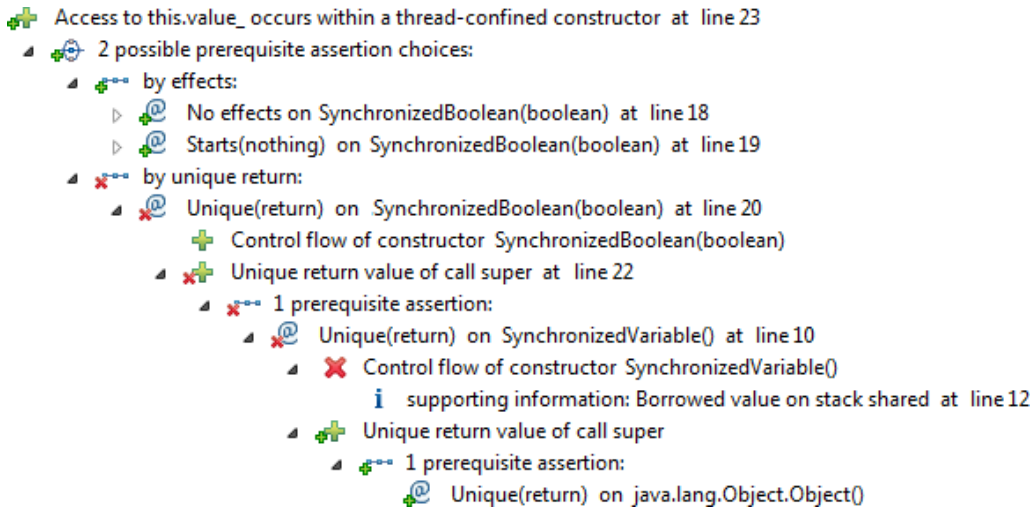


Figure 3.21: A tool view of the (not verified) “by unique return” choice option for the **VarLock** locking model. The view shows a portion of the drop-sea graph decorated with the computed verification results shown in Figure 3.20.

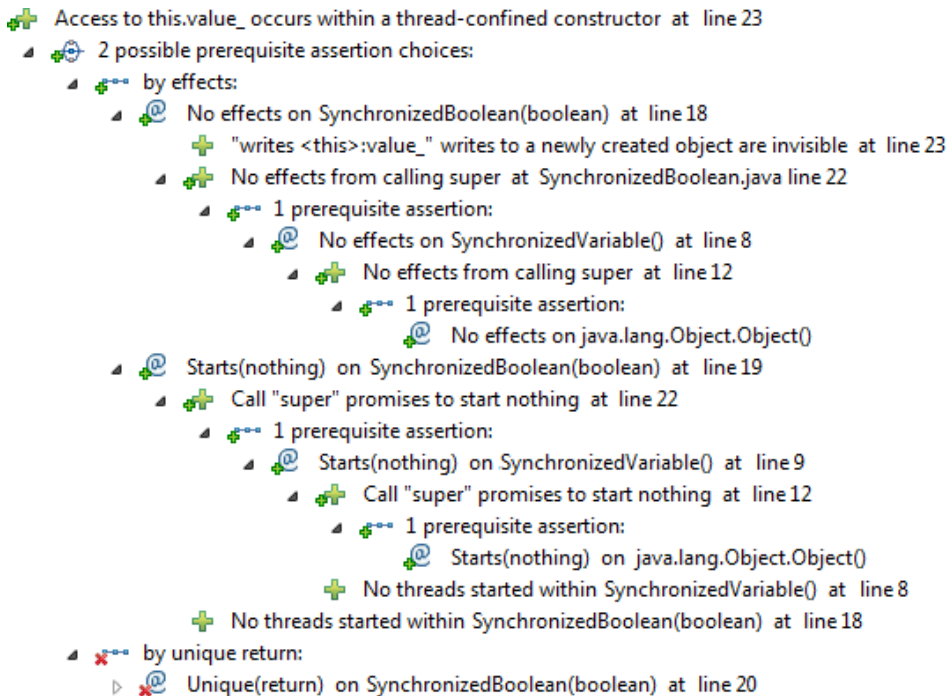


Figure 3.22: A tool view of the (verified) “by effects” choice option for the **VarLock** locking model. The view shows a portion of the drop-sea graph decorated with the computed verification results shown in Figure 3.20.

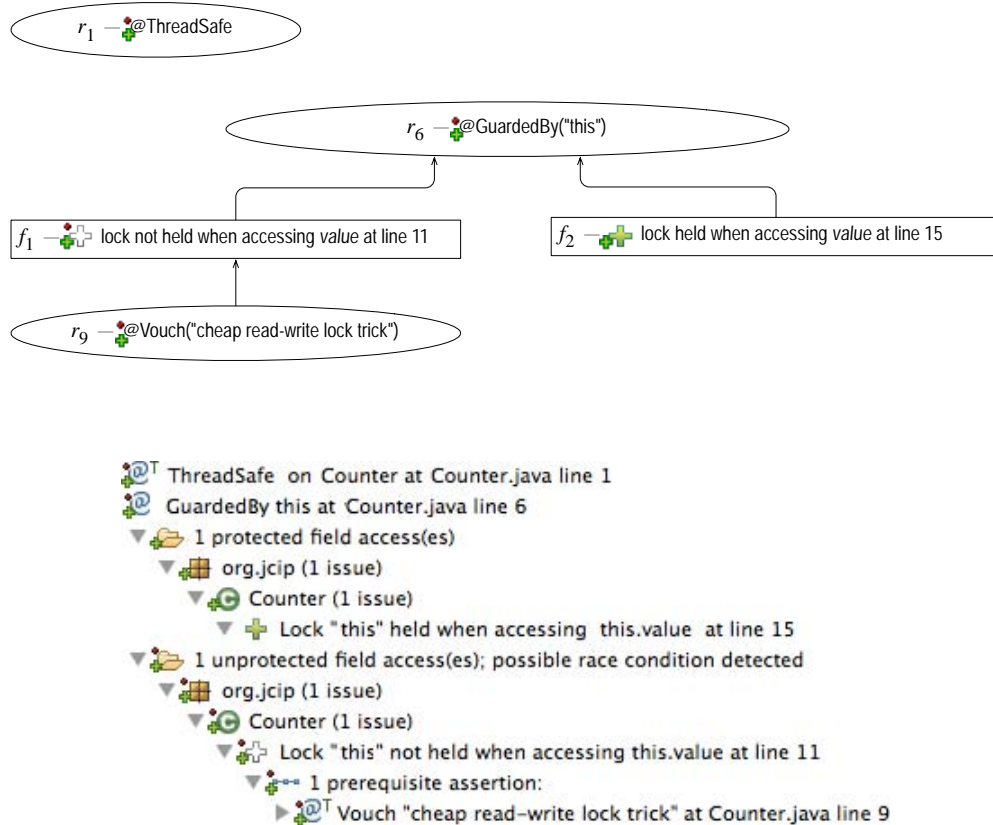


Figure 3.23: (Top) Drop-sea graph for the `Counter` class shown in Figure 3.15 showing computed verification results. A small “+” (to the lower-left) indicates model-code consistency. Because the programmer’s vouch is not verified by analysis, a red dot is introduced above any verification result that depends upon it. The `@ThreadSafe` promise, which is not supported by any analysis results, is “trusted”—a situation that our algorithm highlights with a red dot. (Bottom) A tool view showing the computed verification results. In the tool view a `T` decorator to the upper-right of the `@` icon is used to indicate that a promise is trusted.

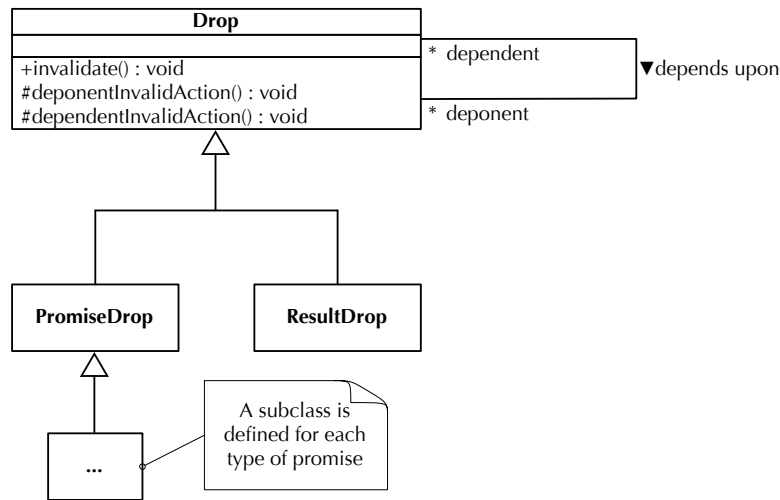


Figure 3.24: UML class diagram for the Drop class.

3.4.6 Truth maintenance

Drop-sea also serves a truth maintenance [102] role. Drops track changes to the software project within Eclipse and thus enable course-grain incrementality for analyses. Drops are implemented as a type hierarchy with the truth maintenance behavior defined within the root **Drop** class. A UML class diagram for the Drop class is shown in Figure 3.24. The key relationship defined is how drops depend upon other drops for the truth of what they represent. A deponent drop gives evidence to support the truth of dependent drops. When any deponent of a drop is invalidated, the default behavior invalidates that drop. Subclasses can override the `deponentInvalidAction` method to change this behavior. By default, when a dependent of a drop is invalidated nothing occurs. Like `deponentInvalidAction`, subclasses can override the `dependentInvalidAction` to change this behavior. The default behavior is rarely changed, but it has been used to avoid invalidating dependent drops that define something that can be referenced by more than one compilation unit, *e.g.*, such as a region or lock name, until all references are gone from the code base.

An example of use of these capabilities within our JSure prototype tool is to clear out drops about a compilation unit when that compilation unit is recompiled. Each compilation unit has a drop constructed for it. All promise drops, result drops, *etc.*, about that compilation unit are setup as dependents of its associated compilation unit drop. When Eclipse recompiles a compilation unit, this drop is invalidated causing all the drops dependent upon it to be invalidated.

Analyses also use the truth maintenance capabilities of drop-sea to perform incremental recomputation, typically at the compilation unit level by linking their result drops to the compilation unit drop that they were examining. Analyses also define drop subclasses with analysis-specific information contained within them. For example, Sutherland attaches binary decision diagrams (BDDs) to each thread coloring promise drop that contains Boolean expressions [103].

3.5 Scoped promises

Scoped promises are promises that act on other promises or analysis results within a static scope of code. In this section we introduce the three types of scoped promises that are supported by JSure: `@Promise` to avoid repetitive user annotation of the same promise over and over again in a class or package and to express the semantics of universality (*e.g.*, all constructors that are ever written for this class—now and in the future—should have this promise), `@Assume` to support team modeling in large systems where programmers are not permitted access to the entire system’s code, and `@Vouch` to quiet overly conservative analysis results. Scoped promises help to “scale up” the ability of a team of programmers to express design intent about a large software system.

3.5.1 Avoiding repetitive annotation

To introduce the `@Promise` scoped promise—our approach to help reduce repetitive annotation by the tool user—we consider the `DateFormatManager` class from version 1.2.8 of Log4j. Log4j is a popular logging package for the Java language maintained as part of the open source Apache project. This class is part of the Log Factor 5 graphical log viewer distributed with this release of Log4j. The `DateFormatManager` class is a simple date formatting class which protects its instance fields by declaring all of its methods to be `synchronized` (*i.e.*, hold a lock on `this`). Its lock policy is declared by the promise: `@RegionLock("Lock is this protects Instance")`. The code, annotated with the minimum number of promises necessary to verify its simple lock policy, is shown in Figure 3.25.

`DateFormatManager` has 8 constructors. We annotate `@Unique("return")` on each of these to verify that the new object remains thread-confined during object construction. All 8 constructors invoke the private method `configure()` to factor out some common code used by the constructors and all the “setter” methods of the class. Figure 3.25 does not elide the source code for the no-argument constructor at line 8 so that this pattern can be observed. The call to `configure()` and the declaration of the method have been underlined to aid the eye. The uniqueness assurance demands that the call to `configure()` also be checked to verify that it does not leak the receiver. Because the uniqueness verification is based upon a modular flow-based program analysis it requires the `@Borrowed("this")` promise on `configure()` at line 37 to verify this property. More precisely, the transitive closure of all methods invoked by the constructors within `DateFormatManager` class must be checked to verify that the uniqueness promise made by the constructors are maintained. Hence, `getTimeZone()` and `getLocale()`, that are invoked directly by `configure()`, must be annotated with `@Borrowed("this")` at lines 23 and 26, respectively.

This can quickly become a tedious exercise for the programmer. The annotation burden on the programmer can be reduced using `@Promise` as shown in Figure 3.26. The `@Promise` scoped promise allows the user to declare—using an aspect-like syntax [69]—the declarations (*e.g.*, types, methods, and fields) within its static scope to duplicate the “payload” promise on. The `@Promises` annotation at line 2 is a vehicle to allow multiple `@Promise` annotations on the same declaration (which is prohibited by the Java language). The annotation `@Promise("@Unique(return) for new(**)")` at line 3 places a `@Unique("return")` annotation on every constructor declared within the class because the target `new(**)` is defined to match any constructor with zero or more parameters. The use of `@Promise` in this case more

```

1  @RegionLock("Lock is this protects Instance")
2  public class DateFormatManager {
3      private TimeZone    _timeZone    = null;
4      private Locale      _locale      = null;
5      private String      _pattern     = null;
6      private DateFormat  _dateFormat = null;
7      @Unique("return")
8      public DateFormatManager() { super(); configure(); }
9      @Unique("return")
10     public DateFormatManager(TimeZone timeZone) {...}
11     @Unique("return")
12     public DateFormatManager(Locale locale) {...}
13     @Unique("return")
14     public DateFormatManager(String pattern) {...}
15     @Unique("return")
16     public DateFormatManager(TimeZone timeZone, Locale locale) {...}
17     @Unique("return")
18     public DateFormatManager(TimeZone timeZone, String pattern) {...}
19     @Unique("return")
20     public DateFormatManager(Locale locale, String pattern) {...}
21     @Unique("return")
22     public DateFormatManager(TimeZone timeZone, Locale locale, ...) {...}
23     @Borrowed("this")
24     public synchronized TimeZone getTimeZone() {...}
25     public synchronized void setTimeZone(TimeZone timeZone) {...}
26     @Borrowed("this")
27     public synchronized Locale getLocale() {...}
28     public synchronized void setLocale(Locale locale) {...}
29     public synchronized String getPattern() {...}
30     public synchronized void setPattern(String pattern) {...}
31     public synchronized DateFormat getDateFormatInstance() {...}
32     public synchronized void setDateFormatInstance(DateFormat ...) {...}
33     public String format(Date date) {...}
34     public String format(Date date, String pattern) {...}
35     public Date parse(String date) throws ParseException {...}
36     public Date parse(String date, String pattern) throws ... {...}
37     @Borrowed("this")
38     private synchronized void configure() {
39         _dateFormat = SimpleDateFormat.getDateTimeInstance(
40             DateFormat.FULL, DateFormat.FULL, getLocale());
41         _dateFormat.setTimeZone(getTimeZone());
42         if (_pattern != null)
43             ((SimpleDateFormat) _dateFormat).applyPattern(_pattern);
44     }
45 }

```

Figure 3.25: Elided code for the `DateFormatManager` class from Log4j after adding the minimum annotations (shown in boxes) required to assure its lock policy.

```

1  @RegionLock("Lock is this protects Instance")
2  @Promises({
3      @Promise("@Unique(return) for new(**)"),
4      @Promise("@Borrowed(this) for getTimeZone() | getLocale() | configure()")
5  })
6  public class DateFormatterManager {
7      private TimeZone    _timeZone    = null;
8      private Locale      _locale      = null;
9      private String      _pattern     = null;
10     private DateFormat  _dateFormat = null;
11     public DateFormatterManager() { super(); configure(); }
12     public DateFormatterManager(TimeZone timeZone) {...}
13     public DateFormatterManager(Locale locale) {...}
14     public DateFormatterManager(String pattern) {...}
15     public DateFormatterManager(TimeZone timeZone, Locale locale) {...}
16     public DateFormatterManager(TimeZone timeZone, String pattern) {...}
17     public DateFormatterManager(Locale locale, String pattern) {...}
18     public DateFormatterManager(TimeZone timeZone, Locale locale, ...) {...}
19     public synchronized TimeZone getTimeZone() {...}
20     public synchronized void setTimeZone(TimeZone timeZone) {...}
21     public synchronized Locale getLocale() {...}
22     public synchronized void setLocale(Locale locale) {...}
23     public synchronized String getPattern() {...}
24     public synchronized void setPattern(String pattern) {...}
25     public synchronized DateFormat getDateFormatInstance() {...}
26     public synchronized void setDateFormatInstance(DateFormat ...) {...}
27     public String format(Date date) {...}
28     public String format(Date date, String pattern) {...}
29     public Date parse(String date) throws ParseException {...}
30     public Date parse(String date, String pattern) throws ... {...}
31     private synchronized void configure() {
32         _dateFormat = SimpleDateFormat.getDateTimeInstance(
33             DateFormat.FULL, DateFormat.FULL, getLocale());
34         _dateFormat.setTimeZone(getTimeZone());
35         if (_pattern != null)
36             ((SimpleDateFormat) _dateFormat).applyPattern(_pattern);
37     }
38 }

```

Figure 3.26: Elided code for the `DateFormatterManager` class from Log4j using the `@Promise` scoped promise to reduce the tedious annotation of its 8 constructors. This reduces the minimum number of annotations required to assure its lock policy from the 12 shown in Figure 3.25 to 4 (shown in boxes).

precisely expresses the programmer’s intent. The programmer wants *all* the constructors to respect the lock policy—both the constructors that exist today and any added in the future.

The annotation `@Promise("@Borrowed(this) for getTimeZone() | getLocale() | configure()")` at line 4 places a `@Borrowed("this")` annotation on the three methods annotated directly in Figure 3.25: `getTimeZone()`, `getLocale()`, and `configure()`.

In practice, the aspect-like syntax is often not needed for `@Promise` because the absence of a `for` clause is defined to mean that the promise is placed on every declaration where it is sensible within the current scope. We could therefore replace the entire `@Promises` annotation shown in Figure 3.26 on lines 2–4 with

```
@Promise("@Borrowed(this)")
```

This promises that no method or constructor in the `DateFormatManager` class will alias the receiver. (As described further in Section A.1.4, the `@Unique("return")` annotation and the `@Borrowed("this")` annotation are defined to be equivalent when placed on constructors.) In essence, with the annotation above, we have changed the “default” with respect to aliasing the receiver within this class.

Using @Promise at the scope of a package

The use of `@Promise` within a `package-info.java` file broadens the scope to the entire package that the file is contained within. The below example shows a `package-info.java` file in the `org.apache.log4j.lf5` package that uses `@Promise` to specify that no threads are started by code code within any class declared in this package.

```
@Promise("@Starts(nothing)")
package org.apache.log4j.lf5;
```

Virtual promises

We refer to promises that are “created” by `@Promise` as *virtual promises*. Similar to how programs on operating systems that support virtual memory do not realize that they are not addressing physical memory, verifying analyses within an analysis-based verification system do not realize that virtual promises are not actually annotated in the code. These promises, from the point of view of the verifying analyses are no different from annotations in the code. In our example from Log4j, no verifying analysis would note a difference between the promises in Figure 3.25 and the virtual promises created by the promises in Figure 3.26. This abstraction, performed by the infrastructure of the analysis-based verification system, simplifies construction of new program analyses.


























The JSure tool allows the user to see the virtual promises constructed by each `@Promise` annotation. Two examples are shown in Figure 3.27 for the `DateFormatManager` class. This user interface helps the programmer confirm that the virtual promises he or she intended were indeed created.

```

2 @Promises({
3   @Promise("@Unique(return) for new(**)"),
4   @Promise("@Borrowed(this) for *(**)")
5 })

```

📁 Scoped promises (2 issues)

- ▲  Promise @Borrowed(this) for *(**) at DateFormatManager.java line 4
 - ▶  Borrowed(this) on DateFormatManager.configure()
 - ▶  Borrowed(this) on DateFormatManager.format(java.util.Date)
 - ▶  Borrowed(this) on DateFormatManager.format(java.util.Date,java.lang.String)
 - ▶  Borrowed(this) on DateFormatManager.getDateFormatInstance()
 - ▶  Borrowed(this) on DateFormatManager.getLocale()
 - ▶  Borrowed(this) on DateFormatManager.getOutputFormat()
 - ▶  Borrowed(this) on DateFormatManager.getPattern()
 - ▶  Borrowed(this) on DateFormatManager.getTimeZone()
 - ▶  Borrowed(this) on DateFormatManager.parse(java.lang.String)
 - ▶  Borrowed(this) on DateFormatManager.parse(java.lang.String,java.lang.String)
 - ▶  Borrowed(this) on DateFormatManager.setDateFormatInstance(java.text.DateFormat)
 - ▶  Borrowed(this) on DateFormatManager.setLocale(java.util.Locale)
 - ▶  Borrowed(this) on DateFormatManager.setOutputFormat(java.lang.String)
 - ▶  Borrowed(this) on DateFormatManager.setPattern(java.lang.String)
 - ▶  Borrowed(this) on DateFormatManager.setTimeZone(java.util.TimeZone)
- ▲  Promise @Unique(return) for new(**) at DateFormatManager.java line 3
 - ▶  Unique(return) on DateFormatManager()
 - ▶  Unique(return) on DateFormatManager(java.lang.String)
 - ▶  Unique(return) on DateFormatManager(java.util.Locale)
 - ▶  Unique(return) on DateFormatManager(java.util.Locale,java.lang.String)
 - ▶  Unique(return) on DateFormatManager(java.util.TimeZone)
 - ▶  Unique(return) on DateFormatManager(java.util.TimeZone,java.lang.String)
 - ▶  Unique(return) on DateFormatManager(java.util.TimeZone,java.util.Locale)
 - ▶  Unique(return) on DateFormatManager(java.util.TimeZone,java.util.Locale,java.lang.String)

```

2 @Promises({
3   @Promise("@Unique(return) for new(**)"),
4   @Promise("@Borrowed(this) for getTimeZone() | getLocale() | configure()")
5 })

```

📁 Scoped promises (2 issues)






- ▲  Promise @Borrowed(this) for configure() | getLocale() | getTimeZone() at line 4
 - ▶  Borrowed(this) on DateFormatManager.configure()
 - ▶  Borrowed(this) on DateFormatManager.getLocale()
 - ▶  Borrowed(this) on DateFormatManager.getTimeZone()
- ▶  Promise @Unique(return) for new(**) at line 3

Figure 3.27: Two tool views showing the virtual promises created within the `DateFormatManager` class from Log4j using `@Promise`. Virtual promises are identified by a `V` decorator to the upper-right of the `@` icon. (Top) All constructors are annotated with `@Unique("return")` and all methods are annotated with `@Borrowed("this")`. (Bottom) All constructors are annotated with `@Unique("return")` (elided) but only three methods are annotated with `@Borrowed("this")`.


```

1 package EDU.oswego.cs.dl.util.concurrent;
2
3 public class Rendezvous implements Barrier {
4
5     @Starts("nothing")
6     @Assume("@Starts(nothing) for new() in IllegalArgumentException")
7     public Rendezvous(int parties, RendezvousFunction function) {
8         if (parties <= 0)
9             throw new IllegalArgumentException();
10        ...
11    }
12    ...
13 }

```

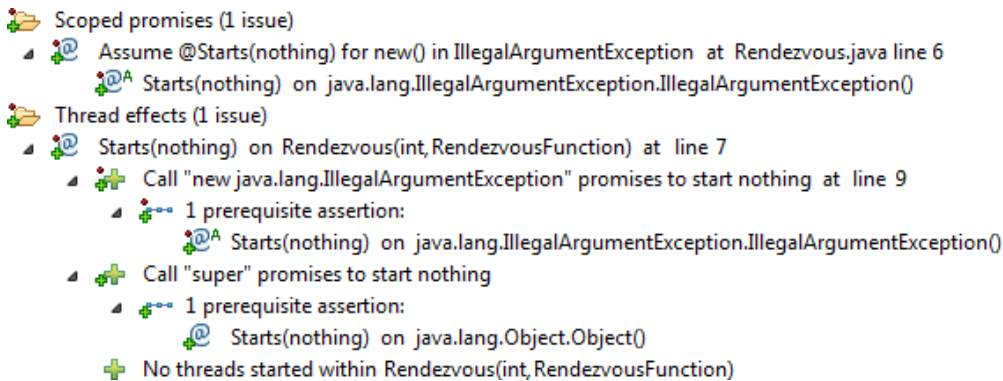


Figure 3.28: An example of using the `@Assume` scoped promise. (Top) Elided code for the `Rendezvous` class from `util.concurrent` which states that the constructor that is shown does not start any threads. (Bottom) JSure screenshot of the results for the verification of the `@Starts("nothing")` promise at line 5. Assumed promises are identified by a `A` decorator to the upper-right of the `@` icon. Because the assumption is not verified by analysis, a red-dot is introduced above any verification result that depends upon it. The red-dot highlights a contingency to the tool user.

3.5.2 Team modeling with assumptions

Real-world software systems are typically decomposed into components and may depend upon many libraries. Thus any modular verification system has to deal with the problem of an assertion within one component requiring, as a prerequisite, an assertion within a second component to be true. A confounding factor, which we now consider, is that these components may be maintained by different programmers due to organizational or contractual boundaries. To help with this problem we introduce the scoped assumption: `@Assume`. The `@Assume` scoped promise allows the user to declare—using an aspect-like syntax [69]—declarations (*e.g.*, types, methods, and fields) outside of its static scope that the user would like the “payload” promise to be consistent for. Virtual promises created by `@Assume` are not checked by verifying analyses. In addition, they are only visible to a verifying analysis when it is examining the compilation unit where the `@Assume` annotation appears (as discussed further below).

Consider the elided code for a constructor of the `Rendezvous` class within `util.concurrent` shown in Figure 3.28. Here, with the `@Starts("nothing")` promise at line 5, the user is stating his or her intent that the constructor implementation start no threads. To verify this

promise, however, the tool requires (as a prerequisite assertion) that the no-argument constructor for `IllegalArgumentException` also promise `@Starts("nothing")`. Because the code for the `IllegalArgumentException` class is outside of the programmer’s scope of interest (*e.g.*, on the other side of an organizational or contractual boundary), the user states, at line 6, this desire and moves on.

Contingent verification results

The use of `@Assume` creates virtual promises, which unlike those created by `@Promise`, are not verified—they are trusted. When an assumed virtual promise is required as a prerequisite assertion for the verification of another promise, then that result is contingent. The result is contingent upon the consistency of the assumed virtual promise.

Virtual promises created by scoped assumptions are indicated in JSure by an A decorator to the upper-right of the @ icon. Because the assumption is not verified by analysis, a red-dot is introduced above any verification result that depends upon it. The red-dot is placed above the verification result decorator (the “+” or “×” at the lower-right). The red-dot alerts the tool user that a contingency exists in the proof for a particular verification result. For example, a red-dot is seen above each verification result shown in Figure 3.28 because they are all contingent upon the assumption made about the `IllegalArgumentException` constructor.

Using @Assume to list desired assertions about other components

Assumptions about a component can become a “to do” for the team working on that component to verify, *i.e.*, to remove the red-dot from the result. Because `@Assume` creates a precise list of desired assertions on other components, we advocate it as a tool to enable design intent modeling among component development teams.

Analysis modularity at the compilation unit level

As noted above, to simplify the construction of new program analyses, analyses are unaware of the existence of scoped promises—a program analysis sees no difference between a normal promise and a virtual promise created by a scoped assumption. However, this raises a limitation of scoped assumptions: *@Assume requires that verifying analyses be modular at the compilation unit level*. To understand the rationale behind this statement we need to discuss the implementation of `@Assume` in the JSure tool.

Each verifying analysis is given one compilation unit to examine at a time by the JSure infrastructure. This compilation unit “focus” is how scoped assumptions are implemented in the tool. Before a compilation unit is passed to an analysis, it is examined by the *promise management* component (shown in Figure 3.7) for `@Assume` annotations. If any exist then the virtual promises that they specify are brought into existence in the forest of ASTs. At this point the forest of ASTs is passed to the analysis. After the analysis completes its examination of the compilation unit and reports its output to the JSure infrastructure, the virtual promises brought into existence by `@Assume` annotations in that compilation unit are erased. The entire process repeats for the next compilation unit to be examined, and so on. An example of this process is shown in Figure 3.29 for two Java compilation units.

	Foo.java	Bar.java
	<pre>@Assume("@Borrowed(this) for doBar() in Bar") public class Foo { public void doFoo() { ... } }</pre>	<pre>@Assume("@Starts(nothing) for doFoo() in Foo") public class Bar { public void doBar() { ... } }</pre>
Analyzing Foo.java	<pre>public class Foo { public void doFoo() { ... } }</pre>	<pre>public class Bar { @Borrowed("this") public void doBar() { ... } }</pre>
Analyzing Bar.java	<pre>public class Foo { @Starts("nothing") public void doFoo() { ... } }</pre>	<pre>public class Bar { public void doBar() { ... } }</pre>

Figure 3.29: An illustration of how virtual promises are created for `@Assume` annotations by the JSure infrastructure. (Top) Elided Java code for two compilation units: `Foo.java` and `Bar.java`. (Middle) The compilation units as seen by program analyses when `Foo.java` is being analyzed. The `@Borrowed("this")` on the method `doBar()` is the result of the `@Assume("@Borrowed(this) for doBar() in Bar")` annotation in `Foo.java`. (Bottom) The compilation units as seen by program analyses when `Bar.java` is being analyzed. The `@Starts("nothing")` on the method `doFoo()` is the result of the `@Assume("@Starts(nothing) for doFoo() in Foo")` annotation in `Bar.java`.

3.5.3 Defining a payload

The general form for `@Promise` is

```
@Promise("payload for target")
```

Similarly, the general form for `@Assume` is

```
@Assume("payload for target")
```

In this section we present the syntax of *payload* portion of both scoped promises. The aspect-like syntax for the *target* portion of both scoped promises is presented in the next section.

The syntax for the *payload* portion is the same as if the payload promise was written in the code except that quotation marks (*i.e.*, `"`) are removed. For example, `@Borrowed("this")` becomes `@Borrowed(this)` when used as the payload of a scoped promise. This minor change in syntax eliminates the need to escape nested quotation marks when expressing scoped promises. Thus, avoiding program text like `@Promise("@Borrowed(\"this\") for *(**)")`.

3.5.4 Defining a target

The syntax used to express the *target* pattern for a scoped promise is shown in Figure 3.30. If the target matches a particular Java declaration then a virtual promise, which is a copy

of the payload promise, is placed on that declaration. The syntax allows matching of type declarations, field declarations, method declarations, or constructor declarations.

The target pattern is realized in a manner somewhat similar to the definition of a pointcut in an aspect-oriented programming language [69]. The difference is that a pointcut specifies a set of points in the program’s logic, *i.e.*, a point of execution, while a scoped promise specifies a set of type, field, method, or constructor declarations (a purely syntactic entity). In the sense that they localize the expression of a crosscutting *specification* concern, scoped promises provide an aspect-oriented specification capability for analysis-based verification.

Our syntax avoids the ambiguity introduced if constituent parts of patterns (*e.g.*, the package name, the type name, *etc.*) are separated by a “.” through the use of the keyword “**in**”. For example, to specify the method `m()` in the type `C` in the package `edu.cmu` we use the pattern “`m() in C in edu.cmu`” rather than “`edu.cmu.C.m()`” because the latter can become ambiguous when wildcards are added into the pattern. The pattern “`m() in C in edu.*`” specifies all methods named `m` taking no parameters declared in a type named `C` in any package under `edu`. The pattern (which we do not allow) “`edu.*.C.m()`” could match the same methods, however, it could also match all methods named `m` taking no parameters declared in a nested type named `C` declared in any type under in the package `edu`.

Several examples of various `@Promise` annotations are shown in Figure 3.31. Several examples of various `@Assume` annotations are shown in Figure 3.32.

3.5.5 Programmer vouches

It is possible for a programmer to vouch for any “ \times ” analysis results reported within a scope of code. This is done with the `@Vouch` scoped promise. The scope of this annotation matches the scope of the declaration where the annotation appears. This means that any “ \times ” result within that scope will be changed to a (hollow grey) “ $+$ ” result. It is used for documentation and quieting overly conservative analysis results. The `@Vouch` scoped promise differs from `@Promise` and `@Assume`: Those promises act on other promises, while `@Vouch` acts on analysis results.

Use of the `@Vouch` annotation is trusted, *i.e.*, it is not verified by analysis. The annotation requires a brief programmer explanation for the vouch being made.

In the example code shown in Figure 3.33 an `init` method is used to set state, perhaps due to an API restriction about using constructors, and then `CentralControl` instances are safely published. A `@Vouch` annotation is used to explain that the `init` method needs to be an exception to the lock policy.

In the example code below a `@Vouch` annotation is used to explain that the `SmokeTest` class is test code that is intended to violate assertions that hold in the rest of the code base.

```
@Vouch("Test code that violates promises about the overall code base")
public class SmokeTest extends ... {
    ...
}
```

<i>Identifier</i>	→	Java Language Specification [52, page 93]
<i>Visibility</i>	→	<code>public</code> <code>protected</code> <code>private</code>
<i>Static</i>	→	<code>[!] static</code>
<i>IdentifierPat</i>	→	<code>[*] { Identifier ∞ * } [*]</code>
<i>QualifiedNamePat</i>	→	<code>{ IdentifierPat * ** } ∞ .</code>
<i>QualifiedName</i>	→	<code>Identifier ∞ .</code>
<i>InNameExp</i>	→	<code>QualifiedNamePat</code> <code>InNameExp</code> <code>InNameExp</code> <code>InNameExp & InNameExp</code> <code>[!] (InNameExp)</code>
<i>InNamePat</i>	→	<code>QualifiedNamePat</code> <code>(InNameExp)</code>
<i>InPackagePat</i>	→	<code>in InNamePat</code>
<i>InTypePat</i>	→	<code>in InNamePat [InPackagePat]</code>
<i>TypeSigPat</i>	→	<code>*</code> <code>{ boolean char byte short int </code> <code>long float double void </code> <code>IdentifierPat QualifiedName } { [] }*</code>
<i>ParameterSigPat</i>	→	<code>{ TypeSigPat ** } ∞ ,</code>
<i>TypeDecPat</i>	→	<code>[Visibility] { QualifiedName IdentifierPat [InPackagePat] }</code>
<i>FieldDecPat</i>	→	<code>[Visibility] [Static] TypeSigPat IdentifierPat [InTypePat]</code>
<i>MethodDecPat</i>	→	<code>[Visibility] [Static]</code> <code>IdentifierPat ([ParameterSigPat]) [InTypePat]</code>
<i>ConstructorDecPat</i>	→	<code>[Visibility] new ([ParameterSigPat]) [InTypePat]</code>
<i>Target</i>	→	<code>TypeDecPat</code> <code>FieldDecPat</code> <code>MethodDecPat</code> <code>ConstructorDecPat</code> <code>Target</code> <code>Target</code> <code>Target & Target</code> <code>[!] (Target)</code>

Figure 3.30: Extended Backus-Naur Form (XBNF) syntax description of scoped promise targets. A concise overview of the XBNF syntax notation is presented in Appendix B.

```
@Promise("@Borrowed(this)") class C {...}
```

Applies to all constructors and methods declared in `C` because the `@Borrowed` promise is only meaningful on constructors and methods.

```
@Promise("@Borrowed(this) for new(**)") class C {...}
```

Applies to all constructors declared in `C`. The `(**)` pattern indicates any number of parameters including zero.

```
@Promise("@Borrowed(this) for !static **(**)") class C {...}
```

Applies to all instance methods and all constructors declared in `C`. The `**(**)` pattern matches both methods and constructors. (The `*(**)` pattern only matches methods, therefore the `**(**)` pattern is shorthand for `*(**) | new(**)`.)

```
@Promise("@Borrowed(this) for !static *(**) | new(**)") class C {...}
```

Equivalent to the promise above but avoids the use of the `**(**)` shorthand.

```
@Promise("@Borrowed(this) for some*(**) & !(someone())") class C {...}
```

Applies to any method declared in `C` with a name starting with `some`, except for a no-argument method named `someone` (if such a method exists).

```
@Region("S") @Promise("@InRegion(S) for * mutable* | int *") class C {...}
```

Applies to fields in `C` declared to be of any type with names starting with `mutable`, or of type `int` with any name. The fields which match become part of the region `S`.

Figure 3.31: Examples of various `@Promise` annotations on a class `C`.

```
@Assume("@Borrowed(this) for new(**) in Foo")
```

Applies the assumption to any constructor in the type `Foo`. The `**` pattern indicates any number of parameters including zero. If there is more than one `Foo` type exists (*e.g.*, in different packages) then the assumption is made for all of them.

```
@Assume("@Borrowed(this) for !static **(**) in * in com.surelogic")
```

Applies the assumption to all instance methods and all constructors declared in any type in the package `com.surelogic`. The `**(**)` pattern matches both methods and constructors.

```
@Assume("@Borrowed(this) for !static *(**) | new(**) in * in com.surelogic")
```

Equivalent to the promise above but avoids the use of the `**(**)` shorthand.

Figure 3.32: Examples of various `@Assume` annotations.

```

1  @Region("private ControlRegion")
2  @RegionLock("ControlLock is lock protects ControlRegion")
3  public class CentralControl {
4
5      private final Object lock = new Object();
6
7      @InRegion("ControlRegion")
8      private String f_id;
9
10     @Vouch("Instances are thread confined until after init(String) is called.")
11     public void init(String id) {
12         f_id = id;
13     }
14
15     public String getId() {
16         synchronized (lock) {
17             return f_id;
18         }
19     }
20
21     public void setId(String value) {
22         synchronized (lock) {
23             f_id = value;
24         }
25     }
26 }

```

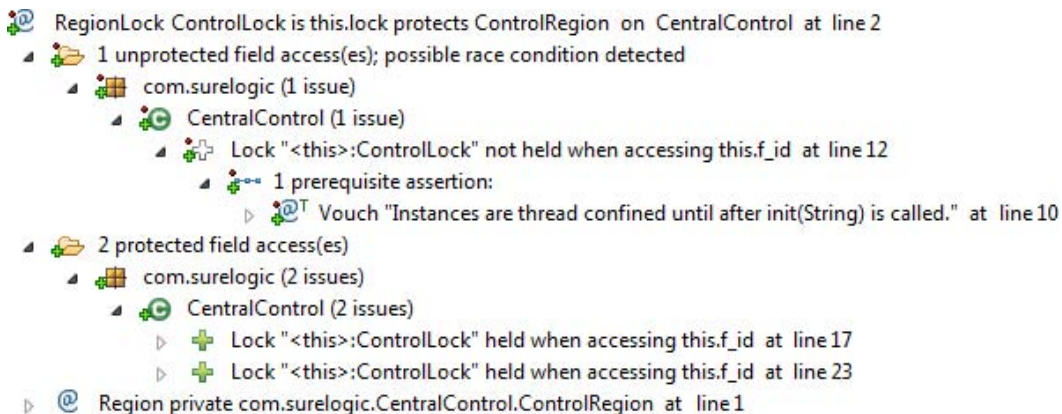


Figure 3.33: An example of using the `@Vouch` scoped promise. (Top) Java code for the `CentralControl` class which vouches that the `init` method will be called while the instance is thread confined and before it is safely published to other threads in the program. (Bottom) JSure screenshot of the results for the verification of the `@RegionLock` promise at line 2. The icon for the “x” analysis result that is within the scope of the `@Vouch`, the access of `f_id` at line 12, is changed from an “x” to a (hollow grey) “+” with the `@Vouch` as a prerequisite assertion. The `@Vouch` promise in the results is identified by a T decorator to the upper-right of the @ icon, indicating that it is trusted. Because the programmer’s vouch is not verified by analysis, a red-dot is introduced above any verification result that depends upon it. The red-dot highlights a contingency to the tool user.

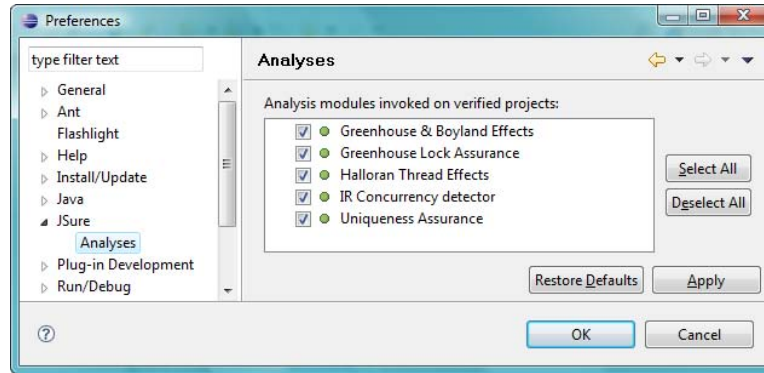


Figure 3.34: Eclipse preferences to turn verifying analyses on and off in the JSure tool.

3.6 Trusted promises

The `@Vouch` is one example of what we call a trusted promise. A *trusted promise* is an assertion that is not verified by analysis.

There are several promises that are never checked by an analysis. An example is the `@ThreadSafe` and `@NotThreadSafe` annotations proposed by Goetz, Peierls, *et al.* in [51]. These annotations are primarily intended for documentation purposes and are not able to be verified by an analysis implemented within the JSure tool. They are nevertheless highly useful to convey design intent, especially to users of the annotated class. For example, consider the `ArrayList` class from the Java standard library to which the `@NotThreadSafe` annotation would apply. It would hypothetically be annotated as shown below to document this aspect of its design intent.

```
package java.util;

@NotThreadSafe
public class ArrayList<E> extends AbstractList<E> implements List<E>, ... {
    ...
}
```

The annotation clarifies the non-tread-safety of this class that might otherwise be (unwisely) assumed to be thread-safe.

The tool also supports turning off a particular analysis. The JSure preference screen to turn verifying analyses on and off is shown in Figure 3.34.

The use of `@Assume` creates, for a particular scope of code on which it appears, a trusted promise. We show assumptions differently in the tool user interface, with an `A` decorator rather than a `T` decorator, as shown above.

3.7 Related work

Our notion of truth maintenance within drop-sea is inspired by AI truth maintenance systems [102]. Further drawing from AI systems, the architecture we present for analysis-based

verification uses a blackboard shared-data style [89, 99] to effectively manage multiple constituent verifying analyses and allow users to understand how the tool reached its conclusions.

Bandera [30] is a system that extracts models from Java source for verification by a model checker and maps verifier outputs back to the original source code. Bandera represents, similar to *drop-sea*, an effort to establish an effective architecture for assurance but is focused on model checking rather than program analysis. Similar to our work, Bandera, and other model checking tools such as SLAM [8], have recognized the need for an effective user experience (*e.g.*, to enable programmer understanding of counter examples found by the model checker).

As in our approach, JML [75], annotates Java programs with special comments that can be used for program verification (or static bug hunting, *e.g.*, ESC/Java [77]). JML does not currently include a mechanism like scoped promises.

The syntax of scoped promises is inspired by AspectJ [69]. However, to the best of our knowledge, use of an aspect-like syntax as part of a specification language is a novel part of our work. The only similar work we are aware of is statically executable advice by Lieberherr, *et al.* in [78] that, lacking any notion of promises, can direct static points for analyses to be executed. Whereas we are specifying locations to place promises, statically executable advice is specifying locations to place bits of analysis code directly into the program.

The FindBugs [62] tool has annotations that, similar to `@Promise`, can annotate declarations within a scope of code⁴. These annotations are: `@DefaultAnnotation`, `@DefaultAnnotationForFields`, `@DefaultAnnotationForMethods`, `@DefaultAnnotationForParameters`. Lacking our aspect-like syntax to express a pattern to match declarations, FindBugs requires several annotations to capture common programmer use cases. The advantage of this approach over ours is that it simplifies the engineering of the tool infrastructure. Further, it may be easier for programmers to understand. But for certain uses, *e.g.*, the scoped promises shown in Figure 1.19 on page 29 used by Sutherland in the Electric case study, it is not expressive enough.

The current code for JSR-305⁵, an ongoing effort to standardize a set of annotations for software defect detection in Java, contains two special-purpose annotations: `@ParametersAreNonnullByDefault` and `@ParametersAreNullableByDefault`. These two annotations are similar to `@Promise` or FindBug's `@DefaultAnnotationForParameters` but can only express intent about the nullability of method parameters. We argue that a general purpose capability to place annotations within a scope of code, such as our approach or the one used by FindBugs, is superior—especially as a standard. The advantage of this approach over a general purpose capability is expediency with regard to its implementation.

3.8 Conclusion

This chapter presents partial solutions to two problems that arise when realizing a practical verification system within a modern IDE: (1) *drop-sea* helps to represent and manage verification results in such a way that a programmer can understand them as changes are made to code and models, and (2) *scoped promises* help to express design intent about large software systems containing multiple components developed by separate teams. We present significant

⁴<http://findbugs.sourceforge.net/manual/annotations.html>

⁵<http://code.google.com/p/jsr-305/>

detail about the design and engineering of the JSure prototype analysis-based verification tool including how the user interacts with the tool.

The JSure prototype tool has evolved based upon feedback from several field trials. These field trials are presented in the next chapter.

Field trials

“Software developers talk a lot about tools.
They evaluate quite a few, buy a fair number, and use practically none.”
— Robert Glass [50]

4.1 Introduction

In this chapter we present the results of nine field trials conducted by the author and other members of the Fluid research group, the research team, with the JSure prototype analysis-based verification tool. The field trials described in this chapter are not formal user studies. There are not control subjects, nor are different aspects of the overall interaction disambiguated. During the time the research team spent in the field its focus was on analysis-based verification and its implementation, not on the users of our prototype tool or their characteristics. We do note, however, that each engagement was conducted with expert professional developers working on complex mission critical code.

The client code examined during each field trial was usually not available in advance, nor was it selected by the Fluid research group. In all cases, the host organization selected the code to be examined as well as the group of engineers who participated. The research team was provided access to the client code upon arrival the first morning. Thus, the field trials did not allow us as controlled an environment as, for example, case studies on real-world code performed at the university. However, the feedback they have provided has been essential to the evolution of our work, in particular, and the work of the entire Fluid project, in general. Proposed promises, scoped promises, and the “red dot” were all, in whole or in part, developed based upon feedback obtained during the time spent in the field using the JSure tool. This explosion of features motivated the formalisms presented in Chapter 2 to ensure that our tool engineering is based upon principled foundations. In this respect, our evaluative use of the field trials, which spanned several years, was both formative and summative.

4.1.1 Organizations visited and code examined

From July 2004 to October 2009 the research team participated in nine field trials of the JSure tool on open source, commercial, and government Java systems. Many of these field trials were performed under strong restrictions relating to disclosure while others were deemed by

Date	Duration (days)	Organization	Software Examined	Code Size (KSLOC)
Jul 2004	3	<i>Company-A</i>	Commercial J2EE Server- <i>A</i>	350
Dec 2004	3	NASA/JPL	Distributed Object Manager	42
			MER Rover Sequence Editor	20
			File Exchange Interface	12
			Space InfeRed Telemetry Facility	18
Feb 2005	3	Sun	Electric – VLSI Design Tool	140
Oct 2005	3	<i>Company-B</i>	Commercial J2EE Server- <i>B</i>	150
Jul 2006	3	Lockheed Martin	Sensor/Tracking (CSATS)	50
			Weapons Control Engagement	30
Dec 2006	1	Lockheed Martin	Equipment Web Portal	75
Mar 2007	3	NASA/JPL	Testbed	65
			Service Provisioning (SPS)	40
			Mission Data Processing (MPCS)	100
			Next-Generation DSN Array	50
Oct 2007	3	NASA/JPL	Maestro	17
			Command GUI	139
			Accountability Services Core	48
Oct 2009	3	Yahoo!	Hadoop HDFS	107
			Hadoop MapReduce	281
			Hadoop ZooKeeper	62

Figure 4.1: Date, duration, organization, software examined, and code size of the Java software systems examined during the 9 field trials of the JSure tool.

the participating company to be more public. We are not permitted to identify the company in two cases: we instead refer to them as *Company-A* and *Company-B*. Figure 4.1 provides an overview of the field trials including the name and size of the client software examined, when the engagement took place, and its duration. A brief description of each client software system examined during a field trial is provided in Figure 4.2.

The corpus of code examined during the field trials represents two major categories of real-world Java software. The first is server/infrastructure software—software that customers of the client organizations use to construct or host enterprise Java applications. The software that falls into this category are the two J2EE servers (from *Company-A* and *Company-B*) and the Hadoop “cloud computing” components. The second major category is naval and aerospace mission support software. These systems support mission data processing and information flow to meet a particular mission objective of the organization that develops and maintains them. Both types of software represent code that is considered important enough that the developing organization was willing to spend time with us to potentially improve its code quality.

In addition to being perceived as important to the client organization, the majority of the client code examined was concurrent. This made the JSure tool, and the predominantly concurrency-oriented mechanical program properties verified by it (listed in Figure 1.3 on page 8), applicable to these systems. The examined software systems were, in all cases,

Software Examined	Description
Commercial J2EE Server-A	The <i>Java Platform, Enterprise Edition</i> is an industry standard for enterprise Java computing. This is one commercial implementation of this standard.
Distributed Object Manager	The Distributed Object Manager (DOM) is a file catalog system which provides a file repository, and a search and retrieval mechanism for mission data.
MER Rover Sequence Editor	The Rover Sequence Editor (RoSE) is used to create command sequences sent to the Mars rovers for every day of the Mars Exploration Rovers (MER) mission.
File Exchange Interface	The File Exchange Interface (FEI) provides a set of command line utilities to manipulate remote data.
Space InfeRed Telemetry Facility	A data high throughput data processing program for the Spitzer Space Telescope.
Electric – VLSI Design Tool	The Electric VLSI Design System is an open-source electronic design automation system.
Commercial J2EE Server-B	The <i>Java Platform, Enterprise Edition</i> is an industry standard for enterprise Java computing. This is another commercial implementation of this standard.
Sensor/Tracking (CSATS)	Common Sensor and Tracking (CSATS) services code written for the U.S. Navy.
Weapons Control Engagement	Weapons Control Engagement code written for the U.S. Navy.
Equipment Web Portal	The Equipment Requirements System code base for Global Combat Support System–Air Force web portal.
Testbed	A distributed hardware system test harness that uses many threads to coordinate software simulations with actual hardware command and control capabilities.
Service Provisioning (SPS)	The Service Provisioning System (SPS) is used for scheduling deep space network services.
Mission Data Processing (MPCS)	A multi-mission Java-based ground system for processing spacecraft data.
Next-Generation DSN Array	An example adaptation of the mission data system to manage an array of tracking antennas.
Maestro	A graphical mission planning tool used by MER to plan science activities.
Command GUI	The user interface to the command uplink system that supports uplink communications with missions.
Accountability Services Core	The Accountability Services Core (ASC) supports telemetry data processing.
Hadoop HDFS	Hadoop Distributed File System (HDFS) is a distributed file system that provides high throughput access to application data.
Hadoop MapReduce	Hadoop MapReduce is a framework for distributed processing of large data sets on compute clusters.
Hadoop ZooKeeper	Hadoop ZooKeeper is a high-performance coordination service for distributed applications.

Figure 4.2: A brief description of the code examined during the nine field trials of the JSure tool.

selected by the client organization, however, the research group did advise that the system should contain significant use of concurrency. In all but one case the advice of the research group was followed. The exception was the J2EE-based Equipment Web Portal for the Global Combat Support System–Air Force (GCSS-AF) system examined in Dec 2006. In two hours 4 models of lock use were annotated into the code by the author working with two client programmers and verified by the JSure tool, covering all lock use in the 75 KSLOC program. In this case the client programmers valued the JSure tool¹—but not as much as developers of the other 19 systems which consisted of more complex concurrent Java software.

4.1.2 Summative evaluation

The results from the field trials are used to provide evidence in support of several claims of this research. These claims, with a summary of the supporting evidence we present in this chapter, are:

- *The JSure prototype tool scales up to use on large real-world software systems.* The empirical evidence is the successful use of the tool on 20 client software systems examined over 25 days during nine field trials. The largest client system examined was 350 KSLOC and the mean size was 90 KSLOC.
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that the tool was effective with respect to defects found.* The quantitative evidence with respect to defects found is that across the nine field trials the JSure tool helped to identify 79 race conditions in 1.6 million lines of real-world Java code—most of which had already passed organizational acceptance evaluation for deployment. The client developers described the difficulty of finding these defects, “It would have been difficult if not impossible to find these issues without [JSure].” and “[JSure] identified logic and programming errors . . . that extensive review and testing did not discover.”
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that they perceive value from the verification results obtained.* The quantitative evidence with respect to verification results is that across the nine field trials we developed 376 models of programmer intent about lock use and were able to verify most of them with the tool by working alongside client programmers. The client developers described the value of the verification results, in particular the value of the tool’s specification language, “[JSure] was reported by all participants as helping them to understand and document the thread interactions that they had already designed and implemented.”
- *Within two or three hours of using the JSure prototype tool to annotate and analyze the client’s code, the behavior and responses of the client developers indicate that they*

¹The JSure results showed the developers that several lock acquisitions could be removed from the code base, thereby improving overall system performance. The GCSS-AF developers liked the verification results obtained from the tool and proposed that analysis-based verification could be used to track down unused “bits” of SQL, stored in Java property files, that are no longer needed by the application, *i.e.*, the property is never referenced in the code base. Keeping property files consistent with the code base during the rapid evolution of their system was difficult for them to manually check. We leave the development of this analysis to future work.

perceive sufficient reward to continue use. The behavioral evidence is continuing to participate in the engagement and inviting other developers to participate. This was observed in 8 of the 9 field trials. The client developers described the value of the immediacy of results, “We found a number of significant issues with just a few hours of work. We really like the iterative approach.”

- *It is feasible to adopt the JSure prototype tool late in the software engineering lifecycle.* The empirical evidence is that 18 of the 20 client software systems examined in the field were in the operations and maintenance phase of the software lifecycle—they had already passed organizational acceptance evaluation for deployment. One of the commercial J2EE servers examined had been in release for 3 years. The verbal evidence consisted of expression by client developers advocating widespread use of the tool throughout the software lifecycle (when code exists), such as, “I can’t think of any of our Java code I wouldn’t want to run this tool on.”

Generally speaking, the field trials show that analysis-based verification is valued by disinterested practitioners—working programmers whose only interest in our work is the immediate value that JSure can potentially provide to them on code they develop and maintain.

As discussed in Section 1.6.2, the field trials provide empirical evidence in support of the contribution of our work as well as the overall vision of the Fluid project. The constituent analyses developed by Greenhouse, Boyland, and Sutherland could not be employed effectively to obtain results meaningful to professional software developers without the contribution of our work. This is because they had no significant support for user interaction, and, more importantly, could not create proof structures that rendered results directly meaningful to developers. For example, the proof structures developed over the course of three days during our first field trial include several thousand nodes in proof trees and several thousand individual underlying constituent analysis results. There is no means for individual developers (or the research team, for that matter) to manage this quantity of separate proof elements without tool assistance.

4.1.3 Formative evaluation

Figure 4.3 shows which of the technical contributions of this thesis were in place for each field trial. (These capabilities are described in Chapter 3.) All nine field trials used the Eclipse-based JSure prototype tool, including the drop-sea proof management system, the @Promise scoped promise, and contingency management via the red dot (*e.g.*, turning analyses on and off, and trusted promises). @Assume and @Vouch were later added to JSure in response to feedback from the field. The last capability added to JSure as part of this work was proposed promises, which was developed in response to feedback from the field to help support model expression (as discussed in Section 1.4.2), but was not available for use during any field trial.

The feedback from the field trials helped to guide our work in two areas: model expression and contingency support. Many client programmers expressed a desire for more help entering annotations into the code. The use of IDE templates for “fill-in-the-blank” annotations (as described in Section 3.2.2) and proposed promises (as described in Section 1.4.2) were developed in the tool address the client programmer’s request for more assistance with model expression. The addition of @Assume and @Vouch (as described in Section 3.5.2 and Section 3.6, respectively) address the request for more approaches to express contingencies.

Date	Drop-sea	Red dot	Scoped promises			Analyses on/off	Trusted Promises
			@Promise	@Assume	@Vouch		
Jul 2004	✓	✓	✓			✓	✓
Dec 2004	✓	✓	✓			✓	✓
Feb 2005	✓	✓	✓			✓	✓
Oct 2005	✓	✓	✓	✓		✓	✓
Jul 2006	✓	✓	✓	✓		✓	✓
Dec 2006	✓	✓	✓	✓		✓	✓
Mar 2007	✓	✓	✓	✓		✓	✓
Oct 2007	✓	✓	✓	✓		✓	✓
Oct 2009	✓	✓	✓	✓	✓	✓	✓

Figure 4.3: Capabilities (developed for this thesis) present during the 9 field trials of the JSure tool.

A note about transition and commercialization

In October of 2006 a commercial company, *SureLogic*, was formed to transition and commercialize the tools and techniques developed by the Fluid research project at Carnegie Mellon University. Several members of the Fluid research project, including the author, are employed by this “spin-off” company and, therefore, SureLogic participated in later field trials—the first being the March 2007 visit to NASA/JPL.

The remainder of this chapter is organized as follows. We begin by describing the methods used to conduct and gather data from each field trial. We then present an analysis and discussion of the field trials focused on the summative evaluation summarized above in Section 4.1.2.

4.2 Methods

In this section we describe the methods used to conduct the field trials of the JSure prototype tool. We start by describing the size and composition of both the research and client teams who participated. Next we present a template agenda for a three-day engagement followed by a description of the client facilities used. We end the section with a discussion of the preparation undertaken by both the research team and the client team for each field trial and the techniques used for data collection.

4.2.1 Participants

The *research team* for each field trial typically consisted of three members of the Fluid project at Carnegie Mellon University. Members of the research team were researchers and engineers that had helped to develop portions of the technology being used. In addition to the author, who participated in all of the field trials excepted the one at Sun in February of 2005, other members of the research team and, in parentheses, the number of field trials in which they participated are: Kevin Bierhoff (1), Nathan Boy (1), Edwin Chan (8), Aaron Greenhouse (2), Larry Maccherone (1), Elissa Newman (3), Dean Sutherland (5).

Date	Organization	# of Engineers: Client Team	# of Engineers: Research Team
Jul 2004	<i>Company-A</i>	8	5
Dec 2004	NASA/JPL	15	4
Feb 2005	Sun	5	3
Oct 2005	<i>Company-B</i>	7	3
Jul 2006	Lockheed Martin	7	3
Dec 2006	Lockheed Martin	2	1
Mar 2007	NASA/JPL	10	3
Oct 2007	NASA/JPL	5	3
Oct 2009	Yahoo!	13	4

Figure 4.4: Engineering team sizes during the 9 field trials of the JSure tool.

The *client team* consisted of engineers from the organization that hosted the field trial. Figure 4.4 reports the size of the client team and the size of the research team for each of the nine field trials. We only count programmers/engineers when reporting the size of the client team. Any members of the hosting organization’s management or administrative staff (*e.g.*, who may have attended the outbrief or issued security badges) are excluded from our count.

The client team participants were, in all cases, the developers and maintainers of the systems that we examined. We were not working with researchers within the client organization that had collected code from one or more engineering groups for examination during the visit. We advocated, with success, to work directly with engineers “in the trenches” when setting up each field trial to have the best chance of getting honest feedback about our approach.

The programmers and software engineers who participated in the field trials were all actively engaged in the development and maintenance of, if commercial, a shipping software product for their company or, if government, mission system software that was in active development and use by the government agency. We did not formally survey the participants, however, we can (via informal discussions) characterize them as experienced—the vast majority with over 5-years working for their respective organizations. We do not view the experience level of client participants as surprising because the task was to examine the source code of software systems important to their organizations (mostly deployed software). Rather, it would be strange for a company or government agency to trust “new hires” with an outside group looking at their code.

We note that working with experienced client developers, to a degree, strengthens the evidence supporting several of the claims listed in Section 4.1.2. In particular, the claims that take into account the “behavior and responses of the client developers.” For example, the time of experienced developers tends to be extremely valuable, and they tend to be very busy and under pressure (when compared to less experienced developers). The fact that members of this group were willing to spend several days working with the research team is impressive—if it was a group of “new hires,” it would be much less impressive, since they may not have had anything very pressing to do anyway. Further, the fact that experienced developers were impressed with issues found through use of the JSure tool is more convincing than if inexperienced developers (who might be impressed easily) had been involved.

4.2.2 Agenda

The template agenda for each field trial is shown in Figure 4.5. The major focus of the engagement, typically a total of 15 hours over the 3 days, is hands-on use of the JSure tool on client code. Later field trials, beginning with the visit to *Company-B* in October 2005, also include hands-on use of the Flashlight dynamic analysis tool.

Each day began with a “meet and greet” and ended with a “wrap-up” primarily focused on ensuring that scheduling of the client team participants and facilities was as planned for the engagement. We purposely limited the introductory talks by both teams—the research team describing the analysis tools to the client team and the client team describing the architecture and design of their code to the research team—to one hour to ensure that the client team was actively using JSure before lunch on the first day. The rationale for this is to avoid busy programmers disappearing over lunch (as is discussed later in this chapter).

The teams separated on the evenings of day one and day two. The client team was encouraged to run the changes made to their code during the day through their automated regression testing suite to see if the changes passed. Some client teams checked-in changes to the code immediately, literally right in front of us, while other client teams tracked code changes for consideration after we departed. The latter group often had no choice but to do this as they were not allowed access to their development network during our visit due to organizational security restrictions.

The research team worked during the evenings of day one and day two to track down and fix any bugs discovered in the analysis tools. This need to patch the analysis tools was due to a constraint imposed on us during most of the field trials: *We were only allowed access to client code in their facilities*. Thus, in most cases, the first time the research team could run the tools on the code that the client had selected was on the morning of the first day. Therefore, if the client’s code exposed a problem in the prototype tool we had to work around it or fix it during the engagement. This constraint was not imposed if the code we examined was open source software, as was the case for the field trials at Sun and Yahoo! (both Electric and Hadoop are open source software).

A *management outbrief* was drafted by the research team on the evening of day two and finalized the morning of day three by all the participants. A larger group from the client organization was invited to attend this outbrief and typically included members of management. The outbrief consisted of a deck of slides that presented the following information about the engagement:

- An overview of the analysis tools, JSure and (sometimes) Flashlight, that were used.
- The names and contact information, *i.e.*, email and phone, for each member of the research and client teams that participated.
- A description of each software system whose code was examined. This description was primarily qualitative, but included a line of code count.
- Quantitative results for each system, *e.g.*, the number of promises annotated in the code (by the client team with the assistance of the research team) and the number of defects that the tools helped to uncover.

Day 1		
Morning:		
<i>Duration</i>	<i>Participants</i>	<i>Activity</i>
10 min	Everyone	Meet and greet
30 min	Everyone	Tool introduction and demonstration presented by the research team
30 min	Everyone	Software system(s) overview, including overall architecture and specific issues in the identified code portion presented by the client team
2 hours	Everyone	Initial tool use on client code
Afternoon:		
3 hours	Everyone	Continue tool use
15 min	Everyone	Wrap-up – confirm day 2 schedule
Evening:		
–	Client team	Regression testing, check-in code modifications
∞	Research team	Writeup findings, fix tool bugs

Day 2		
Morning:		
<i>Duration</i>	<i>Participants</i>	<i>Activity</i>
10 min	Everyone	Meet and greet – check day 2 schedule
3 hours	Everyone	Continue tool use
Afternoon:		
3 hours	Everyone	Continue tool use
15 min	Everyone	Wrap-up – confirm day 3 schedule
Evening:		
–	Client team	Regression testing, check-in code modifications
∞	Research team	Writeup findings, fix tool bugs, start to draft the management outbrief

Day 3		
Morning:		
<i>Duration</i>	<i>Participants</i>	<i>Activity</i>
10 min	Everyone	Meet and greet – confirm time/location of management outbrief (email reminders)
3 hours (all morning)	Everyone Team leads	Finish up tool use Draft the management outbrief with input from all the participants
Afternoon:		
1 hour	Management/ Everyone	Management outbrief and discussion

Figure 4.5: Template agenda used for field trials.



Figure 4.6: Chris Douglas (of Yahoo!) and Nathan Boy (of SureLogic) working inside Yahoo! Building E with the Flashlight and JSure tools on Hadoop MapReduce during the field trial conducted in Sunnyvale, CA on October 28, 29, and 30, 2009. The image quality of this picture is rather poor (it was taken with the author’s cell phone), however it illustrates the conference room configuration used during most of the field trials. Engineers from both teams, working side-by-side on their laptops, running the prototype analysis tools on client-selected code. One laptop is being projected to allow other programmers, including those who just wander by, to observe the ongoing work.

- Qualitative results about the client team participant’s perception of the analysis tools and their value. This included both perceived benefits and perceived limitations.

The management outbrief was the concluding event of each field trial. The formal presentation of the slides was followed by a discussion that included questions from management to the participants, debate about how to improve the content of the slides, and planning for follow-on interaction between the organization and the research team.

4.2.3 Facilities

Each field trial was conducted in the client organization’s facilities. The research group traveled rather than requiring that the client group come to the university. Conducting the engagement at the client’s location helped to allow more of their programmers and engineers to participate.

The work during the engagement was conducted in a common room. The preference was a conference room with a large table that contained, or could accommodate, a projector. The photograph in Figure 4.6 illustrates a typical conference room setup. We worked with the client team to make it as easy as possible for programmers not directly involved with the field trial to come in and observe the proceedings. We were also flexible with regard to the systems we examined during each engagement. If a programmer wandered in, who we met at the cafeteria or who heard about us from a colleague, we often expanded the study to include their code.

4.2.4 Preparation

Prior to the arrival of the research team we asked organizations to ensure that they could build their code in the Eclipse Java IDE. This is because the prototype analysis tools are implemented within Eclipse. This choice was made because Eclipse is the most widely used IDE by Java programmers—more than all other Java IDEs put together [59]. Ensuring ahead of time that client code could be compiled in Eclipse helped us minimize the duration between the start of the field trial and when the tools produced results about the client’s code.

We also asked the client organization to provide a rough line of code count to allow us to estimate the computing requirements for the engagement. The JSure tool requires a large amount of memory that increases linearly with code size. All of the field trials were successfully conducted on computers with two gigabytes of memory (or less). Laptops with two gigabytes of memory are common as of this writing but had to be specially procured by the research group for use on the early field trials.

4.2.5 Gathering data

We gathered data about each field trial through notes and the contents of the slides used for the management outbrief. As described above, the management outbrief slides contain quantitative and qualitative data from each field trial.

4.3 Analysis

In this section we present an analysis of the empirical results from the field trials in support of the following claims of this research:

- *The JSure prototype tool scales up to use on large real-world software systems.*
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that the tool was effective with respect to defects found.*
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that they perceive value from the verification results obtained.*
- *Within two or three hours of using the JSure prototype tool to annotate and analyze the client’s code, the behavior and responses of the client developers indicate that they perceive sufficient reward to continue use.*
- *It is feasible to adopt the JSure prototype tool late in the software engineering lifecycle.*

The tool and the annotations it uses were constantly changing and improving throughout the nine field trials that spanned several years as described above in Section 4.1.3. Where there are important differences between the capabilities of the tool as it exists today and as it existed during a particular field trial we will highlight them in the discussion below.

The JSure prototype tool scales up to use on large real-world software systems

The code sizes reported in Figure 4.1 show that the largest system we examined in the field was 350 KSLOC. This code size is at the limit of what JSure is able to support today without the use of a 64-bit JVM (as discussed below). In July of 2004 analysis of a code base this large on computers with 1GB of memory required extraordinary procedures as the research team reported in its field trial notes:

“What are the barriers to tool use? One is the oddball techniques needed to work at scale. The technique used at [Company-A] was to create a JAR of the entire [Company-A] software. Place that JAR on the Eclipse classpath and only include the source code packages needed by the developer we were working with. Typically, the [Company-A] developer would be next to us on a second laptop with the full code opened so that whole-program queries could be answered (*e.g.*, Do the set of source files that I have loaded contain all the references to this public field?).”

JSure memory use *scales linearly with program size*. The problem the research team encountered in the field is that the computers did not support enough memory to scale with the constant factor imposed by early prototypes of the tool. In this section we discuss several approaches used to reduce this factor and allow the tool to be effective in the field. In summary, we got the memory requirements of the JSure tool below what is already required for development (in any modern IDE), so scalability, with respect to memory use, is limited by the same factor that development limits the size of the code base.

The technique described in the quotation above reduced the memory required by JSure because the tool deals with Java bytecode (*i.e.*, binaries) differently than Java source code. For each type with source code the tool constructs a complete Fluid IR-based Java Abstract Syntax Tree (fAST)². For each type that is only represented in bytecode (*e.g.*, in a JAR) the tool constructs a “stub” fAST that represents the type’s non-private declarations. As a consequence, JSure can represent, in the same amount of memory, far more Java bytecode than Java source code.

The research team did not encounter a code base as large as the one analyzed at Company-A again in the remaining eight field trials. However, the JSure tool no longer requires such unusual techniques to support larger code bases. Several engineering changes to the tool and (external) changes to the environment that it runs in have occurred to help mitigate this problem. These are

- **Smarter eAST-to-fAST conversion:** JSure has to convert the Eclipse representation of each Java type, which we refer to as an Eclipse-based Java Abstract Syntax Tree (eAST), to the Fluid representation, which we refer to as a fAST. We discovered that our approach to conversion (in 2004) held all the eAST structures in the program’s

²The Fluid Internal Representation (Fluid IR) models general purpose data using an enhanced version of the standard ternary representation: unique identifiers, attributes, and values. The enhancements made to the standard ternary representation include (1) ultra-fine-grained tree-structured versioning, (2) abstraction to structured entities such as trees and directed graphs, and (3) persistence. The Fluid IR is used within the JSure tool to represent programs as a “forest” of fASTs, it is also used to model flow graphs of program control flow, bindings from a use to a definition/declaration, annotations of programmer design intent, analysis results, *etc.*

heap until the entire conversion process was completed. This approach caused the memory required for the conversion process to be much larger than what was required to perform analysis. Therefore, on large code bases the tool would run out of memory and fail during eAST-to-fAST conversion. Changing how this conversion was accomplished allowed each eAST structure to be garbage collected as soon as its corresponding fAST was created. This change *cut the memory requirement for the tool in half*.

- **fAST node specialization:** The representation of each node in the fAST was optimized to reduce memory use. This change *cut the memory requirement for the tool by another half*. An example of one specialization that reduced memory use is the elimination of collections that we know will be empty or contain only a single element. For example, leaf nodes in a fAST have no children and, therefore, we can eliminate the empty collection used to reference their children.
- **IR swapping:** JSure was changed to allow portions of its forest of fAST structures to be swapped back and forth from memory to disk. This allows much larger programs to be supported by the tool but can adversely impact analysis performance. This approach is still experimental and may become obsolete because of the migration of most tool users to 64-bit JVMs (discussed below).
- **Arrival of 64-bit Java:** Most Java virtual machines (JVMs) used during the field trials were 32-bit. All 32-bit JVMs that we are aware of strictly limit heap memory to 2GB³. Recently 64-bit JVMs have become more common and these open up much larger heap sizes to the tool. In addition, the typical memory size for a computer used for Java development has increased from 2GB to 8GB.

The use of 64-bit VMs as well as the engineering changes to the prototype tool described above allow JSure to support large code bases. While the prototype tool requires a large amount of memory, we are encouraged that the *tool memory use scales linearly with code size*. Today, the prototype tool comfortably supports code bases up to 150 KSLOC on 32-bit VMs and larger code bases on 64-bit VMs.

The research team encountered few code bases above 200 KSLOC. We hypothesize that this may have to do with scalability limitations in today's widely-used IDEs, such as Eclipse. The experience of the research team during the nine field trials provides support for this hypothesis. For example, above a code size of roughly 150 KSLOC a project in the Eclipse IDE becomes slower and less responsive to the programmer. To avoid this behavior programmers break up their systems into multiple projects that are under this ceiling. Why then, it is reasonable to question, did we encounter projects larger than this ceiling? The largest code base we encountered, the 350 KSLOC J2EE Server at *Company-A*, was not normally developed in Eclipse. Although the programmers used several projects in their normal IDE, for the field trial they put all the code together in one Eclipse project. A second example is the Hadoop project code. The developers of this tool can use Eclipse—but most of them do not. The Hadoop developers use a command-line build tool called *Ant*⁴. Ant is similar to *make* [44] but

³Windows JVM implementations were widely used during the field trials. These further restrict the heap size to the largest continuous area of free memory that the Windows operating system is able to provide to the JVM. This is typically between 1 and 1.5GB, but can be much lower if a driver is loaded in an unusual memory location—a problem we encountered on “tablet” laptops.

⁴<http://ant.apache.org/>

it provides better support for the deep directory structures used by programming languages like Java.

In addition to memory use, the runtime performance of our approach impacts scalability with respect to code size. Performance of the constituent analyses and the infrastructural components (presented in Section 3.3 on page 97) such as drop-sea have been adequate during the field trials with one exception: the *uniqueness analysis*. This analysis was designed and implemented by Boyland in the prototype tool based upon work by Sagiv, Reps, and Wilhelm [98]. The uniqueness analysis verifies the `@Unique` and `@Borrowed` promises in the prototype tool. The algorithm has exponential asymptotic complexity (with respect to the number of edges in the control flow graph for a segment of code). It was believed that performance on real-world code would be reasonable, in a manner similar to Hindley–Milner type inference for the ML programming language [93]. However, we encountered problems in the field with this algorithm as the research team reported in its notes:

“The ability to manage contingencies, *i.e.*, the ‘red dot’, was a critical capability as we had to turn off the uniqueness analysis and [drop-sea] informed us which results were impacted (prior to [drop-sea] this would have been difficult to do).”

The ability of drop-sea to toggle analyses on and off (as described in Section 3.6) allowed the research team to avoid running the uniqueness analysis when it exhibited poor performance in the field. What caused the algorithm to have such poor performance? The Java code we encountered in the field differed from what the analysis authors expected. For example, the research team encountered several methods, contrary to the usual style guidance, that were well over 1,000 lines long. More challenging to the uniqueness analysis, we encountered highly complex control flow, *e.g.*, 10 to 15 levels of nested loops and try-finally blocks. Invariably, these blocks of code would be within a part of the system that the client team deemed critical and wanted to model and assure with JSure. When asked about the code, the client programmers would shrug, tell us they knew the code was a “mess” and indicate that refactoring it was on the list to accomplish (someday).

Our approach enables the replacement of a constituent analysis, and it is planned that the uniqueness analysis will be replaced with a novel permissions analysis based on the work of Boyland, Retert, and Zhao [23, 22]. This analysis has been under development for several years and has the potential to replace and improve the performance of both the uniqueness analysis and the effects analysis in the prototype tool. Another approach (that is less forward looking) is to use the object-oriented effects analysis as a complement to the existing uniqueness (*e.g.*, use it to avoid running the uniqueness analysis if the references that are promised to be unique are not accessed) or add further annotations to the code to reduce the control flow considered by the uniqueness analysis.

At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that the tool was effective with respect to defects found

Across these nine field trials the JSure tool helped to identify 79 race conditions in 1.6 million lines of production Java code. In each system the tool found at least 1 race condition and at most 26 race conditions. Figure 4.7 provides a sense of the programmer effort, in terms

Software Examined	Lock Use Policies	Annotation Count	Annotation Density (Count/KSLOC)
Commercial J2EE Server- <i>A</i>	44	~200	~1.0
Distributed Object Manager	13	54	1.3
MER Rover Sequence Editor	12	50	2.5
File Exchange Interface	4	27	2.3
Space InfeRed Telemetry Facility	3	17	0.9
Electric – VLSI Design Tool	12	61	0.4
Commercial J2EE Server- <i>B</i>	47	~250	~1.7
Sensor/Tracking (CSATS)	55	~200	~4.0
Weapons Control Engagement	8	24	0.8
Equipment Web Portal	4	17	0.2
Testbed	34	98	1.5
Service Provisioning (SPS)	8	21	0.5
Mission Data Processing (MPCS)	50	~200	~2.0
Next-Generation DSN Array	4	17	0.3
Maestro	19	71	4.2
Command GUI	30	154	1.1
Accountability Services Core	13	57	1.2
Hadoop HDFS	5	37	0.3
Hadoop MapReduce	7	30	0.1
Hadoop ZooKeeper	4	18	0.3
<i>Total:</i>	376	1,603	1.0

Figure 4.7: A count of the lock use policies expressed for each system with the corresponding annotation count and annotation density.

of annotation to the code, required to reach that result⁵. Due to disclosure restrictions, we do not break down the identified race conditions by system. On average, the research team identified 1 race condition per day during its engagements in the field. In addition, work with the tool identified several other low-level code issues such as non-final locks (allowing mutation of the lock object as the program executes) as well as higher-level design flaws that were of interest to the client team. The perception of clients, as illustrated by the quotations below, was that they deemed this result to be of value.

“It would have been difficult if not impossible to find these issues without [JSure].”

“The instances uncovered in this analysis were in very mature operational code” [106]

“Held successful Fluid workshop on software for the U.S. Navy under development at ESBA MS2 Moorestown 19–21 July. Team developed 63 lock models and [JSure] identified logic and programming errors in the Common Sensor and Tracking (CSAT) services and Weapons Control Engagement segments that extensive review and testing did not discover.” [72]

⁵Figure 4.7 does not reflect the annotation performed to document thread use policies in Sutherland’s thread coloring approach that was performed on a few of these systems. This information is presented in [103].

The quotations highlight the effectiveness of analysis-based verification to find bugs in concurrent code. All of the systems examined were in production use and had been through organizational quality-assurance except for two: Accountability Services Core and Next-Generation DSN Array were new software actively under development. An empirical result from the field trials is that *traditional testing and inspection fail to identify many concurrency defects*. They remain in the code, even after commercial-level quality assurance is complete, creating reliability and security issues in production systems. We noted a degree of fatalism among programmers about this limitation of their quality assurance practices: Programmers were pleased to identify the concurrency defects in their code, but they were not surprised that they were there.

At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that they perceive value from the verification results obtained

Identification of code defects is not the primary purpose of our approach—it is verification. Across all 9 field trials we developed 376 models of programmer intent about lock use and were able to verify most of them by working alongside client programmers. The client programmers valued the verification results; however, many placed greater value on the documentation of design intent provided by our annotations.

“To me the most valuable thing is the basic fact that you’ve given us a methodology to document the concurrency related design intent. I’m actually considering implementing a policy that you can’t add a `synchronize` to the code without documenting [in JSure annotations] what region it applies to.”

“[JSure] was reported by all participants as helping them to understand and document the thread interactions they had already designed and implemented. This was an unanticipated, and indirect, benefit from the study” [106]

We hypothesize that this is because the value of the verification to developers is largely manifest the *next time they update the code*, since the tool helps them to stay consistent with the documented intent. The field trials were too short in duration to support this hypothesis and we leave it to future work.

Somewhat ironically, the same programmers that valued the annotations once they were in their code did not like to write them. As the research team observed in its notes about the first field trial:

“Annotation syntax was perceived as difficult. Without [Greenhouse] and I at the keyboard our tool would have been pretty much impossible for [Company-A] developers to use.”

This changed dramatically with the introduction of templates to assist the tool user with the annotation syntax (as described in Section 3.2.2). An example of using a template is shown in Figure 4.8. This capability made its debut at Lockheed Martin in July 2006 and allowed the Lockheed Martin developers to use the tool directly, rather than watching the research team operate the tool as had been previously done. This was the first field trial

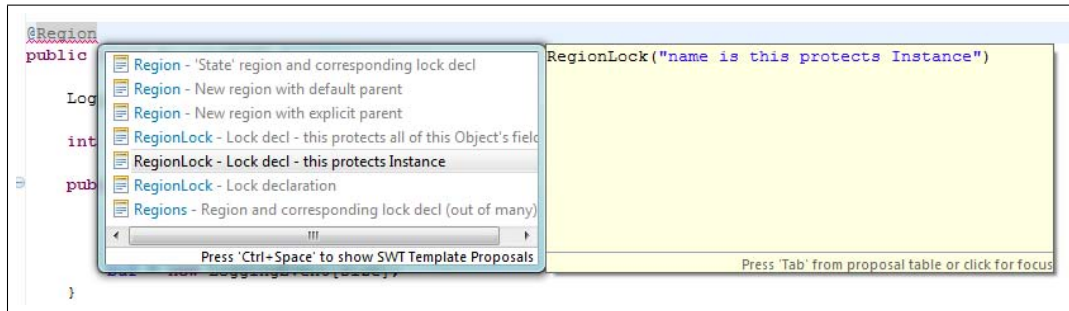


Figure 4.8: Using a template to help with the syntax of the `@RegionLock` annotation assisted many client programmers to use the JSure tool directly with only some oversight by the research team.

where client developers performed most of the tool interaction. Subsequent improvements to the annotation syntax, better documentation of the annotation syntax (meanings and pragmatics), and the use of heuristic techniques to suggest models improved the ability of client developers to perform modeling tasks with the tool.

During the first field trial the desire to get the annotations as a “Java standard” was raised by the client team. Later field trials echoed this feedback primarily, as the quotation below explains, to mitigate the risk to the organization that the effort that they expend annotating their code will not have to be re-accomplished at some later date.

“The annotations need to be standardized.” “Putting these into the [Hadoop] codebase will take a lot of effort (what if we move to or run side-by-side with another tool)” [108]

To use the Java 5 annotations (shown throughout this document) the tool user has to place a library, `promises.jar`, into their classpath to allow annotated code to compile. Prior to 2008 promises were expressed in Javadoc comments and did not require a library⁶.

A topic about which the research team received a great deal of feedback from client developers was the syntax of the annotations. This feedback has resulted in many changes to the annotation syntax originally proposed by Greenhouse in [53] over the years. Figure 4.9 shows an example of the annotation syntax circa 2004 and the same design intent expressed using today’s syntax. The most noticeable syntactic change is the migration from structured Javadoc comments to Java 5 annotations. Changes, such as changing `@lock` to `@RegionLock` or `@mapInto` to `@InRegion`, were made based upon what programmers told the research team “made more sense” to them. In some cases, such as `@synchronized`, annotations have been removed because the intent that they conveyed was redundant. The `@synchronized` annotation was meant to convey that the state of the object under construction remained thread-confined (to the thread that invoked the `new` expression to construct the new instance). The `@synchronized` annotation must always be supported by another annotation, such as `@borrowed this`, to verify what it asserts. Programmers found this syntax to be confusing

⁶Support for Java 5 annotations drove a business decision to release the JSure annotations as open source software under the Apache license at <http://surelogic.com/promises/>. This mitigated the risk to any adopting organization that Carnegie Mellon or SureLogic would “take away” the right to use the JAR containing the annotations. This was a risk that no client organization wanted to take.

and it was changed in 2006 to `@singleThreaded`, and later removed entirely. Another example (not shown in Figure 4.9), deemed horrendous by client programmers, was the annotation `@Aggregate("Instance into Instance")`. This annotation was placed on a field to indicate that the entire state of the object referenced by the field should be aggregated into the state of the object the field is declared within. The much more concise `@Aggregate` has avoided client programmer confusion.

The client programmers that participated in the field trials had never used a tool like JSure before. Is it like testing? Is it like a profiler? The research team found the introductory briefings (presented on the morning of the first day of each field trial) to be an ineffective method of explaining the capabilities of the tool—hands-on tool use was required. The client team engineers quickly grasped the concept of the tool after the research team worked through the specification and verification of one or two lock use policy models with them. After that point the client programmers were able to (1) communicate a focus for modeling by identifying critical code within the system, *e.g.*, “The classes in the `com.kernel` package are more important than the classes in the `com.extras` package, let’s work on `com.kernel` first.” and (2) express concurrency design intent, *e.g.*, “I think we need to put a `@RequiresLock` on that method.”

The ability of drop-sea to present the verification proof in the tool user interface proved valuable in the field, especially to explain verification failures and what possible steps might be taken to eliminate them. The first prototype of drop-sea was implemented just prior to the first field trial. Its capabilities were new to the research team as was observed by the author in the research team’s notes from July 2004:

“The improvement in tool usability gained by being able to interact with the ‘deep’ (*i.e.*, cutpoint spanning) verification result was evident to [Greenhouse] and myself. The [Company-A] folks had never seen or used older tool versions (with their compiler-like output). The results just made sense to them. Especially when trying to figure out how to eliminate a ‘x’.”

Client programmers did not like to have even a single “x” result showing in the tool results. They were willing to use `@Assume` or `@Vouch` annotations to make the result consistent. For example, in one field trial seven `@Assume` annotations were used in 44 locking models (seven out of a total of roughly 200 annotations). The assumptions were used, in all cases, to assume that some assertion about library code held. Overall, the research team observed that client programmers had a strong preference for verified models with a “red dot” over unverifiable models. The research team did not observe rampant abuse of `@Assume` or `@Vouch`, *i.e.*, a client programmer placing them in the code without consideration if the contingency they introduced was reasonable.

The verification results were also used by client programmers to convey to managers the value of the tool. A programmer brought his manager to the computer and showed him a consistent lock model. Drilling into the results the programmer opened the hundreds of field accesses that had been verified—all with a “+”. Scrolling slowly through this list he explained to the manager, “without this tool each one of these has to be tracked down and checked by hand that the right lock is held. If we miss even one mistake then the phone starts to ring.” Subsequently, the research team used this approach in several management outbriefs. It helped to connect the technology with something client developers felt their management could relate to—avoiding trouble calls from customers.

Promise syntax circa 2004

```

/**
 * @region private AircraftState
 * @lock StateLock is stateLock protects AircraftState
 */
public class Aircraft {

    private final Object stateLock = new Object();

    /**
     * @mapInto AircraftState
     */
    private long x, y;

    /**
     * @synchronized
     * @borrowed this
     */
    public Aircraft(long x, long y) {
        this.x = x;
        this.y = y;
    }
    ...
}

```

Promise syntax circa 2010

```

@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public class Aircraft {

    private final Object stateLock = new Object();

    @InRegion("AircraftState")
    private long x, y;

    @Unique("return")
    public Aircraft(long x, long y) {
        this.x = x;
        this.y = y;
    }
    ...
}

```

Figure 4.9: The evolution of annotation syntax based upon feedback from the field trials. (Top) A simple class with lock use policy annotations as they would have appeared in code annotated during an early field trial. (Bottom) A simple class with the same lock use policy annotations expressed in the syntax supported by the JSure tool today.

The focus of each field trial, to use the JSure to document and verify concurrency design intent, was popular with most managers. Some managers expressed concern that the management outbrief would present information critical of the code they were responsible for, *e.g.*, report how many defects were discovered. The ability to claim that JSure helped the engineers find a problem, fix it, then verify the fix, alleviated the fears of a few nervous managers. The focus of analysis-based verification on “good news” (*i.e.*, verification) was popular with management.

During the management outbrief of one field trial, a manager expressed his concern that JSure, and in particular the annotations used by JSure were too complex for most of the programmers who worked in the organization to understand. This resulted in several of the client programmers heatedly explaining that anyone who works on concurrent Java software “had better be able to understand the tool and its annotations, or they are clearly incompetent.”

We suggest that most management skepticism encountered by the research team was rooted in a view that the tool is a potentially disruptive technology. Programmers were enthusiastic because the tool helped them to uncover, in a principled manner, defects that they couldn’t find using other techniques. The fact that the programmers liked the tool and found it valuable was surprising to some managers. Managers sometimes talked about who would pay for the tools, training, and delays caused by the introduction of our approach in their software engineering process.

Within two or three hours of using the JSure prototype tool to annotate and analyze the client’s code, the behavior and responses of the client developers indicate that they perceive sufficient reward to continue use

The idea of the incremental reward principle, a key part of the overall vision of the Fluid project (Section 1.6.1), is that any increment of effort asked of programmers should yield a generally immediate reward—back to the programmers—in the form of added assurance, expression of a model, guidance in evolution, or bug finding. The incremental nature of our approach and its implementation in the prototype tools, as illustrated in the quotation below, was valued by the busy client programmers we worked with in the field.

“We found a number of significant issues with just a few hours of work. We really like the iterative approach. We really like the start-with-nothing approach (We hate tools that spew thousands of problems that are not actionable).” [108]

A threshold of success for the incremental reward principle was passing the *before-lunch test* through the joint effort of the research team and the client developers. For the teams to pass this test, *client programmers had to see useful tool results on their own code before lunch on the first day of the field trial*. If they did not, inevitably, we reasoned, some would disappear over the lunch break as they were drawn into more urgent tasks that just happened to come up. In the nine field trials, there was never evidence of this feared attrition of programmers over lunch on the first day. In fact, in 8 of the 9 field trials the research team experienced the opposite problem: more programmers would return after lunch to work with the research team than the research team started out with in the morning. This led to the use of a projector in the room to allow client programmers who appeared after lunch or wandered in to observe the ongoing work on one of the (several) evaluation computers. In some cases the research team made time to examine client code not originally intended as part of the study.

It is feasible to adopt the JSure prototype tool late in the software engineering lifecycle

All but two of the software systems examined during the field trials were in the operations and maintenance phase of the software engineering lifecycle. (The two exceptions were the Accountability Services Core and the Next-Generation DSN Array—both were new software under active development.) These systems had already passed organizational acceptance evaluation for deployment. Some of the systems were very mature, for example Commercial J2EE Server-*B* had been in release for 3 years. Despite the fact that their code had passed acceptance evaluation for deployment, JSure was deemed valuable by the developers and maintainers of these systems as described above.

There were several impediments to adoption that client developers pointed out to the research team during its time in the field. These are

- **Nightly build support:** The IDE-focus of JSure makes it easy to adopt at the programmer's desktop. However, the ability to run JSure in an automated quality assurance suite, *i.e.*, a “nightly build” [33] is not as well supported. A nightly build typically involves checking out the code from source code control, compiling it into a runnable program, and the execution of a test suite. The ability to run JSure during the nightly build and verify that annotations in the code base remain consistent with the code changes made that day was considered a high priority for several client organizations. This capability is being added to JSure and is planned to be deployed on the Apache Hadoop project. The Yahoo! engineers, in particular, valued this ability and also use a patch process that restricts changes to the code base that cannot pass the automated quality assurance suite.

“We like the idea that JSure can be integrated into our build and run as part of the patch process.” [108]

To a degree, this interest in non-interactive build support, is also motivated by the pain of slow IDE-based analysis (in particular the uniqueness analysis as discussed above). One approach to address this slow interactive performance is tool support for *selective verification* in the IDE. Selective verification would allow the programmer to only ask for the verifying analyses to be run on one or more models of interest, therefore improving interactive analysis performance. These results could be joined with prior non-interactive results in the IDE or fully checked later during the next nightly build. We leave the development of this capability to future work.

- **Help understanding concurrency:** Several client programmers were maintaining software that they did not develop. Either the original programmers had moved on to other jobs/projects or the code had been purchased by the company and subsequently assigned to the client programmer. This situation applied to slightly under half of the client programmers that we worked with. These programmers were very interested in the JSure tools suggestions to get started (as described in Section 3.2.1). These suggestions provide information to the programmer such as where threads and locks are defined or used within the code. The ability of the tool to help them focus in on and understand the concurrency aspects of their software was deemed of great value and, in their opinion, as useful as the ability of the tool to perform verification.

There is a limit to the amount of help JSure is able to provide in this area because it is a static analysis tool. Many of the questions programmers ask about the concurrency aspects of their code cannot be answered (with any precision) by JSure: Is this state actually shared between two threads? I'm worried about the potential of deadlock. Where are two or more locks held in my code?

The identification of this need drove the development of the Flashlight dynamic analysis tool [58] that is briefly discussed below. Starting with the field trial at *Company-B* in October 2005, Flashlight was used as well as JSure.

- **Auto-annotation support:** The research team observed that most programmers were willing to manually type annotations in code (often assisted by templates as discussed above). Many programmers, however, expressed a desire to get further assistance from the tool with annotations that are “obvious” or that are directly asked for by the tool in an analysis result message. These requests led to the development of two auto-annotation capabilities. The first is the “proposed promise” feature that has been presented in previous chapters. The second is the capability of the Flashlight tool to propose lock use policy annotations that can be verified by JSure based upon what Flashlight observed in terms of the program’s locking behavior that is briefly described below.

4.4 Discussion

In this section we discuss our experience in two important areas that are outside of the analysis presented in the previous section. We start with a critique of the prototype tool’s user interface and the user experience that it provides. This is followed by a brief introduction to the Flashlight dynamic analysis tool that was motivated by and subsequently used during our work in the field.

Tool user experience

We have characterized the tool user experience of JSure as utilitarian—it enables the tool user to interact with JSure and accomplish necessary tasks, but it is still far from providing an optimal or elegant user experience. In this section we discuss some of the limitations of our current tool user experience. We leave the design and implementation of approaches to overcome these limitations to future work.

- **Controlling tool analysis execution:** The JSure tool performs its analysis when the user saves changes to any Java source file. This approach is patterned after the Java compiler within the Eclipse IDE. This mode of tool interaction is effective for tool demonstrations and was successful during field trials because it facilitated incremental model expression. For day-to-day use of the tool, especially on larger software systems, it has been criticized by programmers. The reason for this criticism is that JSure adds the analysis to the programmer’s “edit-compile-debug” loop, enlarging it to a “edit-compile-verify-debug” loop. Some programmers are sensitive to any time being added to this loop as it reflects a major component of their everyday work. The unpredictable performance of the tool’s current uniqueness analysis (as discussed above) aggravates the problem.

A separate, but related issue, is the lack of a “Cancel” button for the program analyses. The constituent program analyses were not designed with interactive use in mind. This makes them difficult to control from the user interface. This problem is most noticeable for long running flow-sensitive analyses like the uniqueness analysis.

- **Understanding changes to tool results:** An obvious question posed by the programmer after the analysis executes is, “What changed?” This is an issue when there are more than a small number of models. Some field trials developed several dozen models (as shown in Figure 4.7). For example, 55 models were developed for CSATS at Lockheed Martin. At this scale it sometimes became difficult to manually determine what results had changed as improvements were made to both the models and the code. The *Verification Status* view does not highlight which results changed to the tool user. Therefore, our current tool user interface is not able to directly answer this question.

The selective verification feature proposed above may, in addition to improving interactive performance, help to improve programmer understandability of changing tool results by focusing the user experience on the models that the programmer has deemed interesting.

- **Expression of a focus of interest:** When a programmer is working on the annotation of a particular model the results about other annotations in the code base are considered a distraction. Several programmers have criticized the inability of our current tool user interface to focus on a single promise or the set of promises contained in a class or package.
- **Persistence of results and a baseline:** The tool recomputes its verification results when the Eclipse IDE is restarted. In addition to requesting results persistence, programmers have asked for a degree of control over which results are persisted and compare different sets of results. In particular, the ability to set a “baseline” that can be compared to results on the current code is desired. The “baseline” would represent the last major release of the code or some other past version of interest to the development team.

It is encouraging that despite all the problems with our tool user experience many programmers find JSure valuable enough to continue to use it. This sentiment is captured in the quotation below. It was at the end of a four page “rant” about several of the items described above.

“It sounds like a lot of complaints, but you should know that a lot of helpful data has been gleaned from JSure as well. It’s not all doom and gloom.”

Concurrency-focused dynamic analysis

In this section we discuss the role of dynamic analysis in the context of the work we present in this dissertation. Dynamic analysis is able to improve our ability to perform heuristic promise inference because it is able to observe the running program. While our static approach to “proposed promises” is able to abductively infer what promises are required to complete a partial model, it is unable to propose that the model be created (from nothing). Through observation of the program we have demonstrated the use of dynamic analysis to propose the

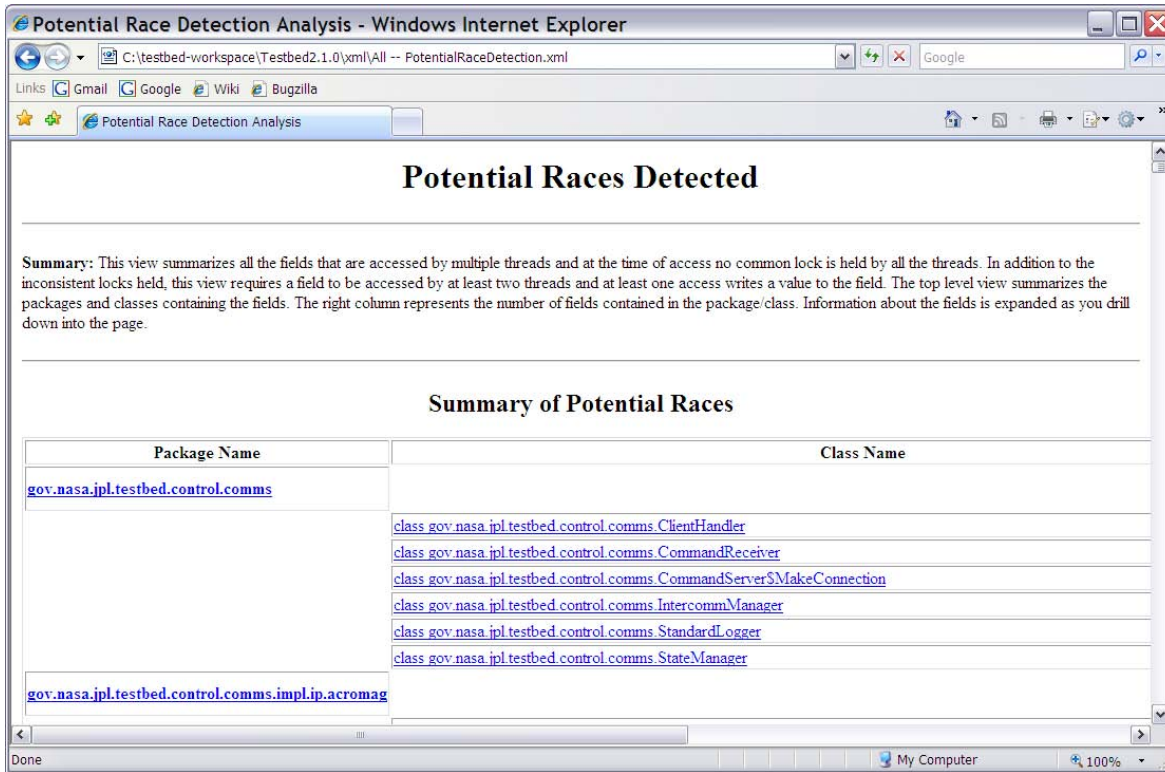


Figure 4.10: Results reported by the AFIT Flashlight tool about potential race conditions that it observed during a run of the Testbed code performed during our field trial in March of 2007. (This screenshot is taken from [106].)

creation lock policy models. Inference of other types of models, *e.g.*, thread coloring [105], are planned as future work.

We first present some background on the development of the **Flashlight** dynamic analysis tool to familiarize the reader with the capabilities and purpose of this prototype tool. We then discuss and evaluate its ability to infer models of lock use policy and evaluate the effectiveness this inference through a comparison of programmer-developed models done during our field trial at Yahoo! on Hadoop HDFS and ZooKeeper.

Background: The **Flashlight** dynamic analysis tool is designed to (1) help programmers better understand the concurrency within their code, (2) uncover concurrency defects (*e.g.*, race conditions and the potential for deadlock), and (3) infer models of design intent that can be verified by JSure. The construction of this tool, as described above, was a direct result of feedback from our work in the field.

Two versions of the **Flashlight** dynamic analysis tool have been created. The first was implemented by the author and Hale at the Air Force Institute of Technology. This version was used in the field between 2005 and 2007 [58]. We refer to this version as **AFIT Flashlight**. The second version, implemented from the ground up at SureLogic, was used at Yahoo! in 2009. We refer to this second version as **SureLogic Flashlight**.

AFIT Flashlight was a rough prototype that reported its output as a series of web pages. Figure 4.10 shows an example of the output from AFIT Flashlight. The capabilities of the

dynamic analysis tool proved popular with programmers despite the difficulty of its use. AFIT Flashlight used Aspect J [69] to perform its instrumentation. This approach, while allowing a prototype to be developed very quickly, required tuning for each Java program instrumented. Working on Aspect J source code in front of client programmers in the field did not inspire confidence in the abilities of AFIT Flashlight. Despite its perception problems, the tool was effective in helping programmers to uncover concurrency defects as reported by Wagner:

“[Flashlight was] particularly helpful to the Testbed developer by finding a likely cause for a troublesome bug that had eluded detection using other tools and methods” [106]

The Testbed developer praised the effectiveness of the Flashlight tool but was critical of its user experience stating that, “it looked like they wrote it on the plane ride out here.”

Based upon the success in the field of AFIT Flashlight a new version of the tool was written at SureLogic. SureLogic Flashlight eliminated the use of Aspect J and provided a flexible query interface within the Eclipse IDE. Figure 4.11 shows the tool displaying the set of fields that were shared between two or more threads. This tool supports a wide variety of queries that allow the tool user to better understand the concurrency that occurs within their programs as well as discover potential defects such as race conditions and the potential for deadlock.

Model inference: AFIT Flashlight was never able to infer lock use policies in a manner that could be understood by JSure. There exists a mismatch between what a dynamic tool, such as Flashlight, can observe and what is required by a static tool, such as JSure. For example, while Flashlight can observe each lock the program acquires it identifies it by a “pointer” to an object in the heap. This information is not usable by JSure. JSure needs to be told which field refers to the lock object, not the address of the lock object itself. This lock identity problem is discussed further by Hale in [58].

SureLogic Flashlight solves the lock identity problem by using a combination of static and dynamic analyses. First, a static analysis is used to create a catalog of every field declared in the program. This analysis is performed when the program is instrumented. A dynamic analysis observes what object is initially assigned to each field and keeps track of this in the statically created catalog. The catalog maps a field declaration to the address of the referenced object. This catalog allows the tool to determine, in many useful cases, which field references an object that is later used as a lock.

This is contemporary work by Greenhouse, Boy, and the author but it is mature enough that we can report some preliminary results. The tool is able to examine the collected data from one or more runs of the program and infer lock policy annotations that can be automatically placed in the code. The user interface is similar to that of proposed promises. The tool interaction to automatically annotate `BoundedFIFO` (used as an example in Chapter 1) from a run of a concurrent logging exercise program is shown in Figure 4.12.

- (Not shown) The programmer runs the logging exercise program using the Flashlight tool to collect data. The data for this run is prepared by the tool for querying and shown in the *Flashlight Runs* view.
- The programmer selects the run in the *Flashlight Runs* view and asks the tool, via a context menu, to infer annotations about the code from the dynamic data.
- The tool previews the automatic edit to the code for the programmer to confirm.

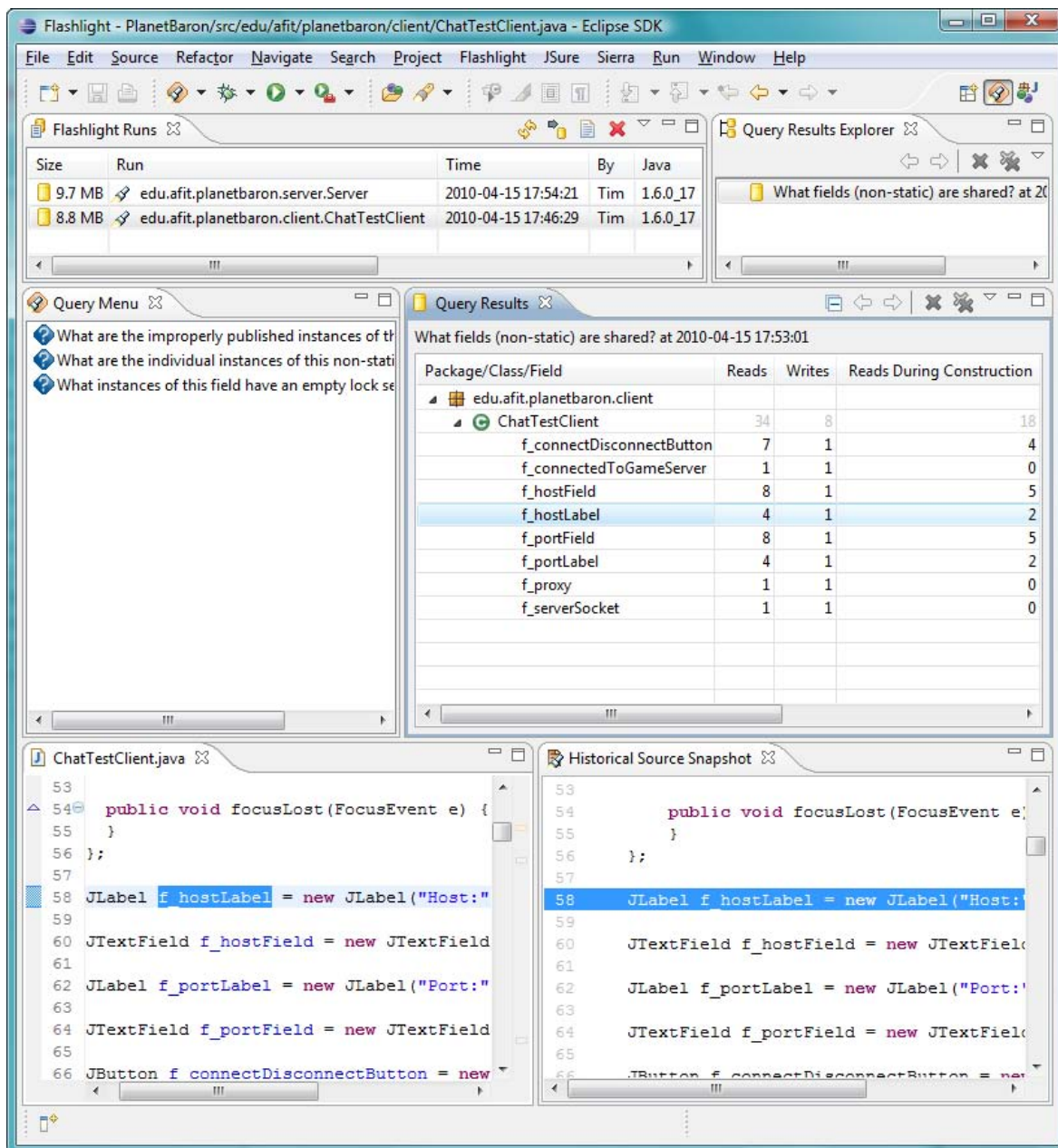
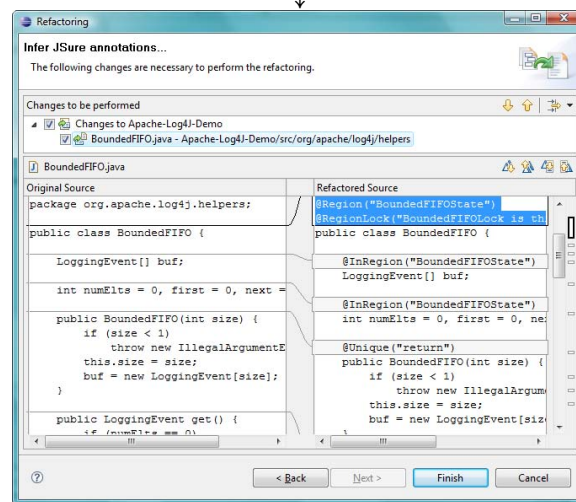
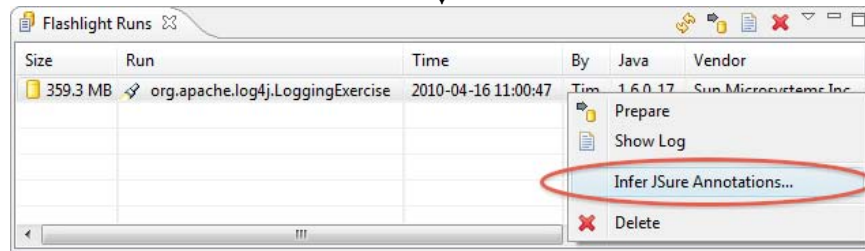
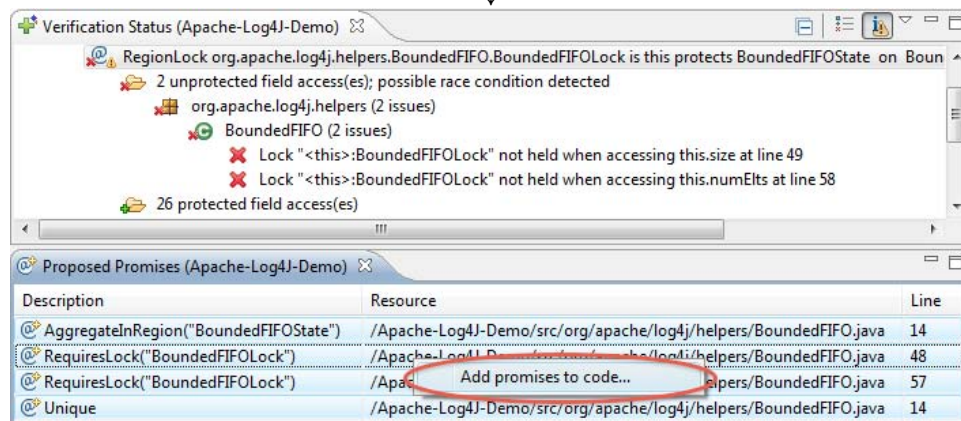


Figure 4.11: The SureLogic Flashlight tool displaying which fields within the `ChatTestClient` program were observed by the tool to be shared between two or more threads. (Top-left) The *Flashlight Runs* view lists the instrumented program runs available for the tool user to query. (Top-right) The *Query Results Explorer* tracks the queries the user has executed and allows them to be called back up in the *Query Results* view. (Middle-left) The *Query Menu* view provides the set of queries that the user can currently run. This view is displaying several “drill-in” queries about the `f_hostLabel` field that is selected in the *Query Results* view. (Middle-right) The *Query Results* view displays the results of the tool user’s queries as a table, a tree, or a tree-table. (Bottom-left) The declaration of the `f_hostLabel` field as it exists in the code now in the Eclipse Java editor. (Bottom-right) The declaration of the `f_hostLabel` field as it existed when the instrumented program run was made.

The programmer selects a program run in Flashlight and asks for the tool to infer JSure annotations using the (circled) context menu



JSure cannot immediately verify the inferred locking model, but (statically, using proposed promises) proposes the “missing” annotations



With the 4 additional annotations in the code, JSure can verify the locking model

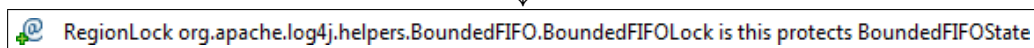


Figure 4.12: Programmer–tool interaction of using Flashlight to infer promises about Log4j’s BoundedFIFO code used as an example in Chapter 1 from a run of a logging exercise program.

Software Examined	Code Size (KSLOC)	Lock use policies (<i>e.g.</i> , <code>@RegionLock</code> promises)				
		At Yahoo!	Inferred	% Overlap	Missed	% Coverage
jEdit	114		3		16	16%
Hadoop HDFS	107	5	17	60%	30	36%
Hadoop ZooKeeper	62	4	16	100%	27	37%
Vuze	359		63		153	29%

Figure 4.13: Evaluation of the lock use policy inference by **SureLogic Flashlight** for four software systems. The number of models inferred by the tool is compared to the number missed to compute a percent coverage metric. The number of policies missed was determined by a count of locks in the source code not covered by an inferred lock use policy. For the two Hadoop systems we compare the number of models developed during our field trial at Yahoo! in October of 2009 (from Figure 4.7) with the inferred models to compute a percent overlap metric. This metric indicates what percentage of the models developed at Yahoo! were inferred by the tool—some models were missed because they were not observed during the program execution used for inference. jEdit is a widely used editor for Java source code (<http://www.jedit.org>) and Vuze is a Java bittorrent client (<http://www.vuze.com>).

- The programmer confirms the automatic edit and the **JSure** tool is invoked to attempt to verify the new annotations. In our example, the lock use model for **BoundedFIFO** is not verifiable. Why? Because two of the methods, `getMaxSize` and `isEmpty`, were never called during the execution of the logging exercise program. Therefore, the inference did not proposed `@RequiresLock` annotations that are needed on these methods. This example highlights a limitation of dynamic inference: Annotations cannot be inferred about code that is not exercised during the observed execution of the program.
- However, the proposed promise feature of our tool picks up where the dynamic inference left off and proposes the missing annotations (in a manner similar to Figure 1.16 on page 26) for the programmer to consider.
- The programmer adds the proposed promises into the code, and with the 4 additional promises now in the code, the **JSure** tool re-runs its analysis and determines that the `@RegionLock` promise is consistent with the code.

The programmers role in the interaction above is that of an auditor—confirming that the intent proposed by the tools faithfully expresses actual design intent.

Figure 4.13 evaluates the inference on four software systems and compares it to the lock use policies developed during our field trial at Yahoo! The level of overlap is encouraging for ZooKeeper with regard to the time this technique could save programmers. The models developed at Yahoo! for HDFS that were missed (the tool found 3 of the 5 models) by **Flashlight** were because that particular code was not exercised during the execution observed by the tool. This is one reason why inference misses models. Another reason is that the lock is not consistently protecting a portion of the program’s state. This could indicate a race condition in the program or it could indicate the use of a hybrid concurrency policy that includes thread-confinement and locking. For example, some code constructs state within a single thread and operates upon that state for a period of time without locks (*e.g.*, during program initialization) then safely publishes that state and subsequently protects it with locking. Finally, if we are not able to determine the field that references a lock object then the inference can not author a lock use policy that is understandable by the current **JSure**

tool. We have not yet categorized which inference failure caused each missed model reported in Figure 4.13. (However, the two models that were expressed in the field about the Hadoop HDFS code but missed by model inference were missed because the code those models express design intent about was not exercised in the program runs used for model inference.)

How could fewer models be missed by inference? *Flashlight* does allow multiple runs to be used as input to model inference, therefore by using more program runs that cover a large amount of the program's behavior fewer models would be missed. The other problems that cause model inference to fail may indicate baroque concurrency policies that are difficult to understand and likely to lead to concurrency defects during system maintenance. These can be investigated through normal use of the *Flashlight* tool (using the interface shown in Figure 4.11).

On average, we developed 6.5 models of lock use policy per day in the field. Based upon our preliminary results, the use of *Flashlight* to infer lock use policies appears to be a promising approach to increase this rate. We caution that determining an actual improvement rate must take programmer auditing of the inferred models into account. Even if the tool can infer 63 models for the Vuze system, the Vuze's programmers have to confirm that the intent proposed in each inferred model faithfully expresses their actual design intent. We leave empirical determination of this improvement rate to be determined in future work.

4.5 Threats to validity

In this section we discuss threats to the validity of the evidence we have presented. That is, are there explanations for the success of the field trials that do not involve meeting our claims or that are unrelated to our work.

We begin by considering that we have only demonstrated the effectiveness and scalability of prior sound analysis work by Greenhouse, Boyland, and Sutherland. We argue to against this view based upon the fact that without the contribution of our work it is doubtful that any of the field trials would have taken place. Given that we were almost always presented with existing code (in fact, already deployed code except in two cases) at large-scale, success in the field required the tools to satisfy scalability with respect to code size and support for adoption late in the software lifecycle on large bodies of existing code. In addition, we organized the engagements in the field for the dual purpose of (1) providing useful results to our host teams and (2) providing useful results for the summative evaluation described herein. This led to unavoidable compromise with respect to uniformity of process and engagement structure. Nonetheless there were important commonalities in the engagements, such as the agenda structure and style of interaction between experimenters and the developer partners. The agenda structure and the fixed limited intense interaction of the *in situ* engagements resulted in the need to address effectiveness with respect to facilitation and repair of defects found and other aspects of perceived value of the approach by practicing programmers and compatibility with the incremental reward principle. In addition, as described in Section 4.1.3, all of the field trials involved use of versions of the evolving drop-sea infrastructure and interaction support for developer users.

The constituent analyses developed by Greenhouse, Boyland, and Sutherland could not be employed effectively to obtain results meaningful to professional software developers without these interventions. This is because they had no significant support for user interaction, and,

more importantly, could not create proof structures that rendered results directly meaningful to developers. For example, the proof structures developed with *Company-A* over the course of three days include 2,695 nodes in proof trees and 2,146 individual underlying constituent analysis results. The proof structures developed in the thread coloring case study performed by Sutherland on Electric include 24K nodes in proof trees and 12K individual underlying constituent analysis results [103]. There is no means for individual developers (or the research team, for that matter) to manage this quantity of separate proof elements without tool assistance.

We now consider that the deep knowledge of Java technology and practices that the research team brought to each field trial was largely responsible for the success the research team observed in the field, *i.e.*, our approach and the prototype tools based upon it were largely irrelevant. We argue against this view based upon the continued tool use the research team observed during the field trials. When the tools uncovered a design flaw in a system being examined, the participants typically moved to a whiteboard to discuss the problem and work to develop potential solutions, however, they soon returned to hands-on use of the tools. The prototype tools were in use by the participants for the majority of each field trial. However, the expertise of the research team cannot be wholly discounted. The research team did, in many cases, help client programmers improve their knowledge about how to engineer safe concurrent Java code. Two areas that were often discussed were (1) the Java memory model [52] and (2) effective use of the `util.concurrent` library [51]. Therefore, while there was value to the client team from the ability to consult with the research team, we suggest that this did not overshadow the value of our approach and the prototype tools based upon it. Further, in several instances the client engineers were more knowledgeable about Java technology and practices than any member of the research team.

We now consider the *external validity* of our evidence beyond the corpus of code examined and the organizations visited during our field trials. We caution against generalizing our results too far beyond the circumstances under which we have demonstrated a degree of success in the field. We worked with six different development organizations. All six of these organizations have a culture that promotes the importance of software quality. Failures in these systems result in mission impact or angry phone calls from customers—both of which contribute to higher operating costs or, worse, mission failure. Developers in these organizations are highly motivated to eliminate bugs. It is not clear if developers in other organizations with different priorities would be willing to invest the same level of effort. The software we examined tended to have very stringent quality requirements and a high cost of failure. It is not clear if organizations developing other types of software, *e.g.*, web apps, would be as willing to invest the effort or be as interested in the results.

4.6 Conclusion

This chapter presents the results of nine field trials we conducted with the JSure prototype analysis-based verification tool. Generally speaking, the field trials show that analysis-based verification is valued by disinterested practitioners—working programmers whose only interest in our work is the immediate value that our tool can potentially provide to them on code they develop and maintain.

The next chapter presents two case studies of adding new aggregate analyses to JSure.

Case studies: Constructing aggregate analyses

“If a listener nods his head when you’re explaining your program, wake him up.”
— Alan Perlis

5.1 Introduction

In this chapter we present two case studies of adding new aggregate analyses to the JSure prototype tool: *thread coloring* and *static layers*. These case studies are used to evaluate the verification framework (*e.g.*, drop-sea, scoped promises, proposed promises, the red dot) developed as part of this work and implemented in the JSure tool with respect to the benefit the framework provides for the incorporation of new analyses. The questions we consider in this chapter include the following: Is the framework we provide considered to be useful by analysis authors? What capabilities provided by the framework are the most useful? What capabilities could be improved? What is missing?

Thread coloring was developed by Sutherland and the technical details of his approach are presented in [103]. Static layers was done by the author and the technical details are presented in the following chapter. Experiences from these two case studies with the verification framework developed as part of our work are summarized in this chapter. Generally speaking, the case studies show that our work helps to facilitate the addition of new sound analyses into the JSure prototype tool by providing a verification framework that simplifies implementation and by providing capabilities, such as `@Promise`, that are widely needed. Figure 5.1 summarizes the perceived benefits and limitations of capabilities provided by our verification framework from the point of view of an analysis author.

5.2 Case study: Thread coloring

Sutherland, using our approach to analysis-based verification, has developed an approach, called *thread coloring*, that allows developers to document their thread usage policies in a manner that enables the use of sound scalable analysis to assess consistency of policy and

Framework capability	Described in Section	Benefits (+) and limitations (–)
Reporting interface	2.3 & 3.4	(+) Simple—one or two lines of code per result
Promise management	3.3	(+) “Scrubs” out promises that are not well-formed (–) Provides little help parsing annotations that have a non-trivial syntax
Truth maintenance	3.4.6	(+) Enables incremental recomputation of results (+) Provides a blackboard to hold partial (or local) analysis results
Results reporting UI	3.2.3	(+) Presents the structure of the verification proof (–) Inability to view the proof “in reverse” (–) Inactive during reanalysis
Proposed promises	1.4.2 & 2.2	(+) Avoids describing a “missing” annotation in the result description (as text) (+) Simple—one or two lines of code to propose a promise
@Promise	3.5.1	(+) Reduces annotation density (+) Takes advantage of stylized naming schemes at many APIs (–) Largest scope of code it can be applied to is a Java package (–) Aspect-like syntax
@Assume	3.5.2	(+) Requires analyses to be modular at the compilation unit level (encourages composition) (–) Aspect-like syntax
@Vouch	3.5.5	(+) Analyses do not have to verify unusual—but correct—coding idioms (<i>e.g.</i> , the cheap read-write trick shown in Section 3.4.4)

Figure 5.1: A summary of the perceived benefits and limitations of several capabilities provided by the verification framework implemented in the JSure prototype tool from the point of view of an analysis author. Many of the framework capabilities listed are provided by the drop-sea proof management system (as described in Chapter 3).

as-written code [103]. Thread coloring is a useful technique to statically verify concurrency policies that do not involve locking, such as the thread-confinement policy adopted by most object-oriented GUI frameworks (*e.g.*, SWING/AWT and SWT).

In this section we report on the strengths and weaknesses of our verification framework with regard to its support of Sutherland’s thread coloring. We start by discussing the effectiveness of our `@Promise` scoped promise to reduce the number of annotations required to document thread usage policies. We then discuss how drop-sea simplified the implementation of verifying analysis for thread coloring.

@Promise is effective for documenting thread usage policies

In Sutherland’s approach, *colors* are notionally associated with one or more program threads, or more precisely give names to the particular roles that threads take on in an executing program. Each color is given a user-meaningful name, *e.g.*, AWT or *Worker*. A *color constraint* annotated on a method or constructor constrains which threads are allowed to invoke that code to threads of a compatible role. For example, a constructor annotated with the color constraint `@Color("AWT")` should only be called within the AWT thread or a method annotated with the color constraint `@Color("Worker")` should only be called from a thread that has taken on the *Worker* role.

Thread coloring verifies an aspect of the program’s control flow, therefore, its analysis requires that all of a program’s methods and constructors have a color constraint to be statically verifiable. Sutherland reduced this annotation burden on the programmer using two techniques: color constraint inference and use of `@Promise`. In the context of this work we consider the effectiveness of the latter.

Sutherland reports that the stylized naming schemes at many APIs, *e.g.*, where accessor methods are prefixed with *get* or *is*, allowed effective use of the `@Promise` scoped promise to significantly reduce a programmer’s annotation burden for their own code. The technique also enabled Sutherland to express the thread usage policies of Java libraries.

“We use scoped promises primarily to reduce annotation density for API methods. The API of Electric’s Database contains 1,731 methods that are referenced from other modules. By using scoped promises, we replace over 1,700 color constraint annotations with six scoped promises in each of the nine packages in the Database module. Similar use of scoped promises in the remainder of Electric saves an additional 331 annotations. Thus, 888 annotations remain to be written after the use of scoped promises, for an annotation density of ~6.3 per KLOC.” [105]

Therefore, our `@Promise` annotation, leveraging stylized naming schemes at many APIs is highly effective for reducing the expression cost, in terms of annotations, for thread usage policies.

A limitation of our current implementation of `@Promise` is that the largest scope of code that it can be applied to is a Java package. Sutherland’s quotation above alludes to the impact of this limitation—he had to duplicate the same six `@Promise` annotations in nine packages. Sutherland proposes that `@Promise` annotations be allowed on static modules defined in a module system, the *Fluid module system*, he proposes in [103]. We view this proposal to be consistent with the “enter-once” principle that motivated our design of `@Promise`: If

the programmer is expressing a unitary concept, then its expression should not have to be scattered throughout the code.

Drop-sea simplifies analysis implementation

Prototype implementations of drop-sea were used by Sutherland to implement the sound static analysis used to verify the programmer design intent expressed in his approach. In addition to the result reporting and proof management capabilities, Sutherland took significant advantage of the truth maintenance system (TMS) [102] features provided by drop-sea. He states

“We use the TMS to support reporting, as a blackboard for holding partial (or local) results from our analyses, and as an engine for proving global consistency of results. In addition, the TMS provides simple support for incremental analysis.” [103]

By “simple” Sutherland is referring to the ability for him to use the TMS capabilities provided by the **Drop** class to systematically avoid re-analysis of the entire Java program when only a small set of Java compilation units has been modified. Sutherland’s analysis also uses drop-sea to store intermediate analysis results and help to manage the worklist he uses to drive color constraint inference and his analysis. He concludes that drop-sea was a strong enabling technology for portions of his analysis implementation:

“The drop-sea truth maintenance system by Halloran, along with its canonical use in the Fluid system, enabled my experiments with incremental recomputation of results.” [103]

Sutherland notes a limitation of the JSure tool user interface when presenting verification results for thread coloring. The *Verification Status* view (shown in Figure 3.5 on page 95) allows programmers to examine the structure of the verification proof—but only in one direction. The view allows the tool user to follow *dependent* results and promises, it does not provide a view that allows the tool user to follow *deponent* results and promises¹. This inability to view the proof “in reverse” is a limitation in thread coloring when the user is trying to determine why a color constraint on a method or constructor has been found inconsistent. In principle, we agree with these criticisms and plan to address them in future work.

Sutherland’s implementation benefited significantly from the use of our approach, in particular the use of drop-sea and scoped promises. We now consider a second case study, performed by the author, of adding support for the specification and verification of a new mechanical program property to our prototype tool which is not concurrency-focused: static layers.

5.3 Case study: Static layers

The technical details of our approach to the specification and verification of static layers are presented in the next chapter. Prior to presenting these technical details we evaluate

¹These relationships are shown in Figure 3.24 on page 117.

the effectiveness of our verification framework toward the addition of this new specialized analysis.

We undertook this case study to (1) evaluate the benefit of our framework with respect to the addition of new analysis attributes and (2) show that our approach was not only useful for the verification of concurrency-focused attributes.

Drop-sea facilitated the implementation of this specialized analysis in one week

The entire implementation for this case study took roughly one week of programmer effort. This implementation time does not include the design of the specification language, which took significantly more time to conceptualize and document. The implementation was simplified through use of the result reporting and proof management capabilities of drop-sea. This allowed implementation efforts to focus on the new specification language provided to document static program structure design intent and its verifying analysis rather than how to report results to the user or the details of how to construct a program-level verification result (*e.g.*, check each model and then combine the results).

One observed weakness of our framework is that it provides little help for parsing program annotations that have a non-trivial syntax. Several of these are used in our approach to the specification and verification of static program structure. For example the following annotation defines a set of types that includes all of the types contained within the `java.util` package except for the `Enumeration`, `Hashtable`, and `Vector` types.

```
@TypeSet("UTIL=java.util & !(java.util.{Enumeration, Hashtable, Vector})")
```

This type set definition, the programmer named `UTIL`, is used by other annotations.

Because the specification language we designed for static structure is very complex, roughly half of the implementation time was spent implementing the parser to support the new promises using the ANTLR parser generator².

@Promise avoids the implementation of “one off” imitators

Our approach to the specification and verification of static program structure includes the notion of a static layer (in an implementation). Our approach allows the programmer to document what is allowed to (statically) reference a layer and what the layer itself is allowed to reference. The technical details are presented in the following chapter, however, here we discuss how `@Promise` simplified the specification of what classes compose a layer. The original specification language design contained the following annotation on a package:

```
@InLayer("MODEL")  
package edu.afit.smallworld.model;
```

Normally, the `@InLayer` annotation appears with a type declaration and is used to include that type in a layer. The use of an `@InLayer` annotation on a package was originally a shortcut to indicate that all the classes declared in that package were to be included in the layer. This shortcut was later deemed to be unnecessary due to the ability of `@Promise` to

²<http://wwwantlr.org/>

accomplish the same effect, placing a particular `@InLayer` promise on all the types declared in a package, without the cost, in terms of programmer effort, to implement a “one off” imitator of `@Promise`. Therefore, the shortcut annotation shown above is expressed in our prototype tool as follows:

```
@Promise("@InLayer(MODEL)")  
package edu.ait.smallworld.model;
```

Avoiding annotations that duplicate, for a single promise, the capabilities of `@Promise` is desirable to *reduce implementation effort* when adding a new specialized analysis to our prototype tool. Is this always the correct decision with regard to facilitating programmer understanding of our specification language? Is `@Promise("@InLayer(MODEL)")` on a package clearer to tool users than `@InLayer("MODEL")`? We leave these questions to be answered in future work.

5.4 Conclusion

This chapter presents two case studies of adding specialized analyses to support the verification of new program attributes, *thread coloring* and *static layers*. These case studies are used to evaluate the benefits and limitations of the verification framework implemented in the JSure prototype tool as part of our work. The case studies show that our work helps to facilitate the addition of new sound analyses into the JSure prototype tool by providing a verification framework that simplifies implementation and by providing capabilities, such as `@Promise`, that are widely needed. The limitations identified are left to be addressed in future work.

The next chapter presents a detailed technical description of our novel approach to the specification and verification of static program structure.

Specification and verification of static program structure

“The higher your structure is to be, the deeper must be its foundation.”
— Saint Augustine

6.1 Introduction

Software does not need structure—one set of bits that can execute on a machine is just about as good as any other. We impose structure on software to allow us to better understand it and control its complexity. Software has several different structures that co-exist, *e.g.*, module structure (static), component-and-connector structures (dynamic), and allocation structure (to one or more external environments) [12]. Our focus is on the module structure, or *static structure*, of a software system. We propose a lightweight specification technique to express the structure of an object-oriented system motivated by the *uses* relationship but using the *references* relationship as a useful approximation. Previous work by Murphy and Notkin [85] has demonstrated the utility to practicing programmers of tool support to understand and maintain structural models of their code. We build upon this work by supporting composition of multiple, possibly overlapping, structural models and supporting the specification of layered systems. We have designed sound static analyses to verify model-code consistency for the Java programming language and implemented them within the JSure prototype tool.

When a programmer examines a module within a software system several questions come to mind: What does this unit do? What other software units does this unit use? What other software units is this unit allowed to use? Our approach helps the programmer to answer the latter two questions, but is primarily concerned with the last. The relationship of interest to the programmer is *uses* originally defined by Parnas in the mid-1970’s [92]. We adopt the following (more modern, but essentially equivalent) definition by Bass, Clements and Kazman:

“One unit *uses* another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be extracted.” [12]

In popular object-oriented programming languages, *e.g.*, Java, it is not possible to statically determine what a unit, *e.g.*, a class or package, is using. Therefore, we adopt the *references* relationship as an unsound and incomplete—but useful—approximation. For our purposes a “unit” will be a Java type. Thus, to answer the Java programmer’s question, “What other software types is this type allowed to reference?” we need a precise specification of the set of types that any particular type is allowed to reference. The consistency of this specification with the code can be verified using a straightforward static analysis.

Our approach is inspired by the demonstrated utility of *reflexion models* proposed by Murphy and Notkin [85] to specify a structural model of the source code for Microsoft Excel. Our approach combines the creation of the high-level model and a mapping from the high-level model to the source code together via source code annotations—primarily a syntactical difference—but our purpose is the same: to help programmers express, understand, and maintain the static structure of their code. Our primary contribution to prior work is the addition of a lightweight approach to specify and verify static layers with well defined semantics that we believe are consistent with traditional layered semantics. In addition, our approach more naturally facilitates composition of multiple overlapping static models.

The driving hypothesis of our work is that non-trivial software systems have multiple models of structural design intent that overlay the source code in manner that is almost never a simple partition of the module structure defined by the programming language. One structural model might express intent about a layered architecture. A second might restrict use of a costly commercial library to specific portions of the code. A third might ensure that the constructor of a class is only invoked from a factory method within a second class (that has, perhaps, been assigned the responsibility to manage instances of the first class). Our intent is to allow the programmer to precisely specify these models of structural design intent.

This work is presented, in the context of this thesis, as evidence that sound combined analyses for analysis-based verification is scalable with respect to new assurance attributes, in particular non-concurrency related program attributes.

Section 6.2 uses a small Java program, SmallWorld, to motivate and present our specification approach. We introduce our approach to specifying layers within a system, precisely define its semantics, and prove that our approach is consistent with the traditional software engineering definition of a layered system. Section 6.3 presents the analyses used to automatically verify model-code consistency using the JSure analysis-based verification tool.

6.2 Specification

To introduce our specification approach we use an interactive adventure game, called SmallWorld. This software is used in Software Engineering classes at the Air Force Institute of Technology to introduce iterative methods of software engineering and design patterns. SmallWorld uses the model/view/controller pattern [71] to concretely demonstrate, via a series of lab exercises, the benefits of a layered architecture to the students. The Java packages which make up SmallWorld are briefly described below.

- `edu.afit.smallworld.model` contains classes that represent game places (*e.g.*, rooms), items, characters, and the player. All these entities are aggregated into instances of a

World class that represents the world the player is within. The classes in this package make up the domain layer of the game.

- `edu.ait.smallworld.controller` contains game logic that operates on the game world. It also controls the loading and saving of games using the persistence package.
- `edu.ait.smallworld.persistence` contains logic to save and load a game world to and from an XML file. This logic is separated from the model to isolate the complexity of encoding and decoding XML.
- `edu.ait.smallworld.ui` contains two text-oriented user interfaces. This is in fact a package tree that contains several sub-packages. The first user interface interface allows the user to interact with the game by typing text commands within a console window. The second user interface allows the user to interact with the game by typing text commands within an AWT/SWING window. Both user interfaces share a single text parser.

In addition to these packages a large suite of JUnit [14] tests have been created to unit test the software.

6.2.1 A simple model

We begin with a simple model of the static structure of the SmallWorld controller, model, and persistence packages which is shown in Figure 6.1. We will improve upon this model in the next section. The model is made up of several `@MayReferTo` promises about the program. The argument to this promise specifies the complete set of Java types that the target of the promise is *allowed to reference*—or statically depend upon. This corresponds to a promise that the target will compile and run if the specified set of types is a subset of the set of types on the program's classpath. We refer to the specified set of types with the term *type set*.

The target of a `@MayReferTo` promise must be a type. If the `@Promise` scoped promise is used to make this promise target a package then it specifies the complete type set that all classes and interfaces declared in that package—not including sub-packages—are allowed to reference. The type set implicitly includes the targeted type itself as well as the types declared in the package `java.lang`. The promise at the top of Figure 6.1 targets (using `@Promise`) all the types declared in the `edu.ait.smallworld.model` package and specifies that they are only allowed to reference each other and all the types declared within the `java.util` package. The `package-info.java` file was added to the Java language to allow annotations and Javadoc on a package. Our promises take the form of Java annotations. The second promise in Figure 6.1 targets (using `@Promise`) all the types declared `edu.ait.smallworld.persistence` package. In this case the type set specification is more complex:

```
edu.ait.smallworld.{model, persistence} | org.jdom+ | java.{io, net, util}
```

This type set includes all the types declared in the SmallWorld model package, all the types declared in the SmallWorld persistence package, all the types declared within the `org.jdom` package and its sub-packages (indicated by the `+` suffix), and all the types declared within the `java.io`, `java.net`, `java.util`, and (implicitly) `java.lang` packages. In general, type sets of

package-info.java within the edu.afit.smallworld.model package:

```
@Promise("@MayReferTo(edu.afit.smallworld.model | java.util)")
package edu.afit.smallworld.model;
```

package-info.java within the edu.afit.smallworld.persistence package:

```
@Promise("@MayReferTo(edu.afit.smallworld.{model, persistence} | org.jdom+ |"
    +" java.{io, net, util})")
package edu.afit.smallworld.persistence;
```

WorldController.java within the edu.afit.smallworld.controller package:

```
package edu.afit.smallworld.controller;

@MayReferTo("edu.afit.smallworld.{model, persistence} | java.io.File")
public final class WorldController { ... }
```

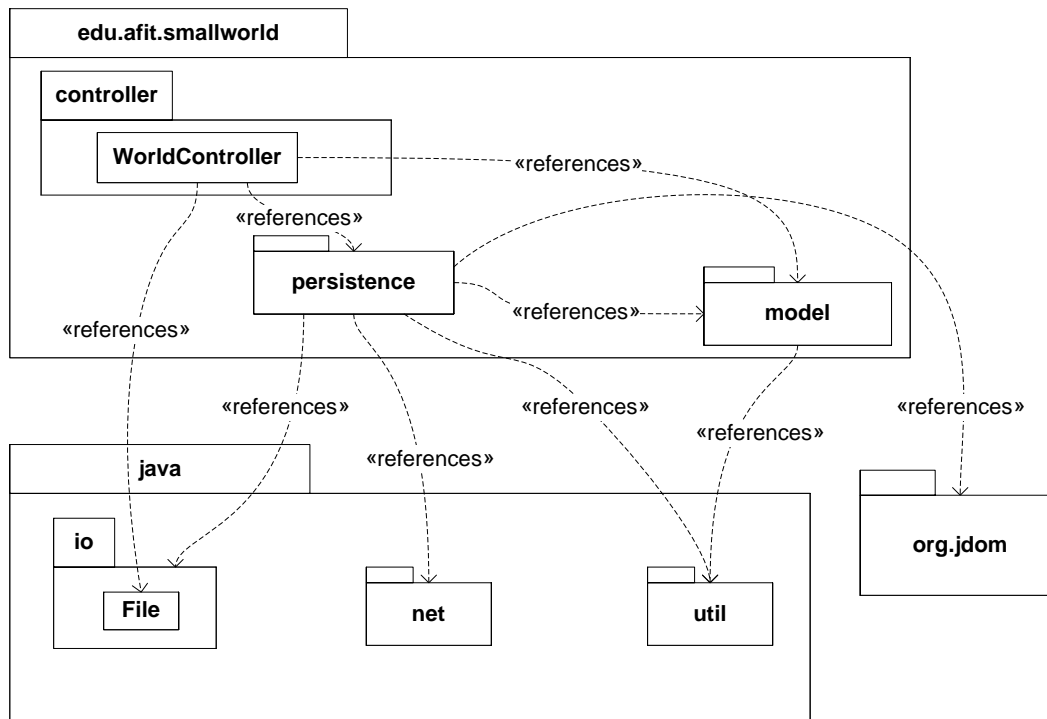


Figure 6.1: A low-level model of the static structure of the SmallWorld controller, persistence, and model packages. The promises added to the SmallWorld code are shown above a UML diagram illustrating their semantics.

arbitrary complexity can be formed using the following syntax: `|` for union, `&` for intersection, and `!` for set complement. Another example type set specification (from Figure 6.2) is

```
java.util & !(java.util.{Enumeration, Hashtable, Vector})
```

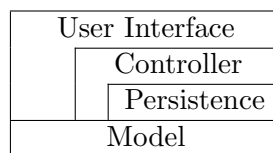
This specifies all the types declared within the `java.util` package except `Enumeration`, `Hashtable`, and `Vector`—perhaps used to prohibit the use of the older Java 1.0 collections. This typeset also implicitly includes all the types declared in `java.lang`.

If the `@MayReferTo` promise targets a type, *e.g.*, the `WorldController` class in Figure 6.1, then it specifies the complete type set that type is allowed to use. As noted above, the type set implicitly includes the targeted type itself as well as the types declared in the package `java.lang`.

What are the semantics if a `@MayReferTo` promise appears both on a type and, through the use of `@Promise`, that type’s package? If a type set, C_p , is specified on a package and a type set, C_t , is specified on a type within the package then that type is allowed to reference the type set $C_p \cap C_t$ (*i.e.*, the intersection of the two sets). In addition, $C_t \subseteq C_p$ must hold or the promises are nonsensical (*i.e.*, a type can’t expand the type set promised by its package). The verifying analysis views this situation as two assertions on the type that are each individually verified. The first placed by the `@Promise` and the second by direct annotation.

6.2.2 A layered model

The model in Figure 6.1 is a utilitarian expression of design intent about `SmallWorld`’s static structure but, as its associated tangle of a UML diagram illustrates, it is not a clear expression of what the designer had in mind. He or she might sketch something similar to



which better conveys that `SmallWorld` is intended to be a *layered system*—where lower layers do not have unrestricted access to higher layers. While sketches like this are common in practice and, indeed, evocative of designer intent, they have some problems when used as a specification for implementation and evolution. This particular sketch

- is not precise about how the program’s modules (*i.e.*, packages and classes) are mapped into the specified layers (*e.g.*, do both `SmallWorld` user interfaces map into the “User Interface” layer or do they each get their own “User Interface” layer?),
- is silent about what libraries (with potentially large associated license fees) are allowed to be used by each layer, and
- is easy for a busy programmer (over time) to forget.

We propose the model in Figure 6.2 as an improvement over our utilitarian model in Figure 6.1 in the sense that it is closer to our designer’s sketch while at the same time it does

package-info.java within the edu.afit.smallworld package:

```
@TypeSets({
    @TypeSet("UTIL=java.util & !(java.util.{Enumeration, Hashtable, Vector})",
    @TypeSet("XML =org.jdom+ | UTIL | java.{io, net}")
})
@Layers({
    @Layer("MODEL      may refer to UTIL"),
    @Layer("PERSISTENCE may refer to MODEL | XML"),
    @Layer("CONTROLLER may refer to MODEL | PERSISTENCE | java.io.File")
})
package edu.afit.smallworld;
```

package-info.java within the edu.afit.smallworld.model package:

```
@Promise("@InLayer(edu.afit.smallworld.MODEL)")
package edu.afit.smallworld.model;
```

package-info.java within the edu.afit.smallworld.persistence package:

```
@Promise("@InLayer(edu.afit.smallworld.PERSISTENCE)")
package edu.afit.smallworld.persistence;
```

WorldController.java within the edu.afit.smallworld.controller package:

```
package edu.afit.smallworld.controller;

@InLayer("edu.afit.smallworld.CONTROLLER")
public final class WorldController { ... }
```

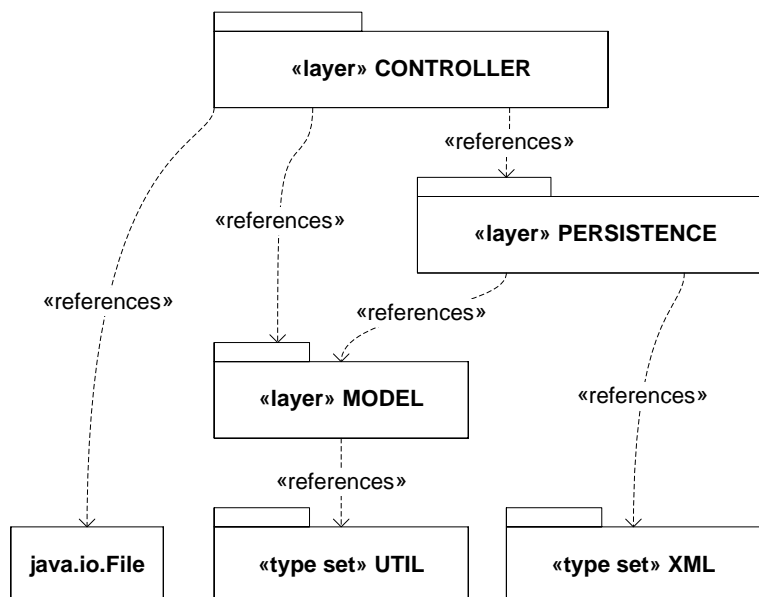


Figure 6.2: A more abstract model than Figure 6.1 of the static structure of the SmallWorld controller, persistence, and model packages. The promises added to the SmallWorld code are shown above a UML diagram illustrating their semantics.

not suffer from the sketch's implementation and evolution problems (we delay consideration of the user interface layer until the next section). Of course, the disadvantage of our model over the sketch is its higher expression cost. We now describe the elements that make up the model in Figure 6.2.

TypeSet definitions

Figure 6.2 introduces the `@TypeSet` annotation to define and name a set of types within the code. A named type set adds no additional semantics—it is simply a definitional mechanism. A named type set may replace a type set specification or be used within it. For example, the promise on the persistence package in Figure 6.1 could have used the equivalent form

```
@TypeSet("JDOM = org.jdom+")
@Promise("@MayReferTo(edu.afit.smallworld.{model, persistence} | JDOM |"
        +" java.{io, net, util})")
package edu.afit.smallworld.persistence;
```

A type set definition is allowed to define itself in terms of other type sets. One example, shown in Figure 6.2, is the use of the type set `UTIL` to define the type set `XML`. A type set definition may also define itself in terms of one or more layers (described below).

Type set definitions may only be placed on a package. They have global visibility and may be fully qualified by prefixing their name with the name of the package where they are annotated. For example, the fully-qualified name of the `JDOM` type set defined in the example above is `edu.afit.smallworld.persistence.JDOM`. References to the type set from annotations on types declared in the same package do not need to be fully qualified. If multiple type set definitions are specified on a package then the `@TypeSets` annotation is used like the example in Figure 6.2.

Layer definitions

Figure 6.2 uses the `@Layer` annotation to declare three layers within the `SmallWorld` game: `MODEL`, `PERSISTENCE`, and `CONTROLLER`. A layer declaration names the layer and specifies the type set that types mapped into the layer are allowed to reference. The declaration does not specify the set of types that comprise that layer—layers are empty at declaration. The `@InLayer` annotation is used to populate the contents of a layer and may only be placed on a type declaration. The annotation indicates that the specified layer now includes that type. Types are allowed to be mapped into more than one layer, however, we postpone describing the semantics of this until the next section.

`@InLayer` annotations may only map types into a layer if those types are located within the package tree where that layer is declared. This restriction ensures it is possible to determine the complete set of types comprising a layer by examining the `@InLayer` annotations within the package where the layer is declared and all its sub-packages.

Layer declarations may only be placed on a package. Similar to type set definitions they have global visibility and may be fully qualified by prefixing their name with the name of the package where they are annotated. References to the layer from annotations on types declared in the same package do not need to be fully qualified.

What if a type is mapped into a layer and a `@MayReferTo` promise appears on that type and/or, through the use of `@Promise`, that type's package? Is this allowed? What does it mean? Yes, this situation is allowed and can be given reasonable semantics. If a type set, C_p , is specified in a `@MayReferTo` on a package and a second type set, C_t , is specified in a `@MayReferTo` on a type within that package that is mapped into a layer that may refer to the type set, C_l , then the type is allowed to reference the type set $C_p \cap C_t \cap C_l$. In addition, $C_t \subseteq C_p$ and $C_t \subseteq C_l$ must hold or the promises are nonsensical (*i.e.*, a type can't expand the type set promised by its package or the layer it is within). In the case that the entire package, through the use of `@Promise`, is mapped into the layer then $C_p \subseteq C_l$ must hold as well. The verifying analysis views this situation as three assertions on the type that are each individually verified.

Specification of layers implies that there exists an ordering such that lower layers may not depend on higher layers. This ordering clearly exists in the model shown in Figure 6.2. We delay formally describing the details of this constraint until after the next section about composition of layered models. We simply note, informally, that our approach to layers is consistent with their traditional semantics.

6.2.3 Composing layered models

In this section we describe the ability of our technique to compose multiple layer models over a single software system. Figure 6.3 extends the model in Figure 6.2 to include the game's two user interfaces. In this model, SmallWorld's `textui` and `textui.parser` packages are mapped into both the `CONSOLE_UI` and `SWING_UI` layers. This is illustrated in the UML diagram at the bottom of Figure 6.3.¹

Figure 6.3 is not the only approach to model the static structure of the SmallWorld user interface. We acknowledge that many readers may find the previous model shown in Figure 6.2 adequate for SmallWorld (*i.e.*, the top layer can use anything it wishes). This highlights the incremental nature of our approach—it provides incremental benefits (in the form of model-code consistency) for incremental modeling efforts (in the form of promises). We present the more complex model in Figure 6.3 primarily as an example to illustrate the capability of our approach to compose multiple layer models.

The semantics of a type being mapped into multiple layers is straightforward. If a type, t , is mapped into layers L_1, L_2, \dots, L_n where $n \geq 0$ with associated allowed to reference type sets C_1, C_2, \dots, C_n then the allowed to reference type set, C , for t is

$$C = \bigcap_{1}^n C_n.$$

Further, if t has a `@MayReferTo` promise and/or its enclosing package, through the use of `@Promise`, has a `@MayReferTo` promise, with allowed to reference sets C_t and C_p , respectively, then the allowed to reference set for t is $C \cap C_t \cap C_p$. As noted in the previous section, $C_t \subseteq C_p$ and $C_t \subseteq C$ must hold or the promises are nonsensical. Finally, $C_p \subseteq C_i$ where $1 \leq i \leq n$ must hold for each i where the entire package enclosing t is mapped into L_i .

¹We have taken some liberties with UML semantics in Figure 6.3. Specifically, UML prohibits including an element within more than a single package—even a package with our «layer» stereotype. This limitation of UML to describe layers has also noted by the authors of [29].

package-info.java within the edu.afit.smallworld.textui package:

```
@TypeSets({
    @TypeSet("SWING = java.awt+ | javax.swing+"),
    @TypeSet("IO    = java.io")
})
@Layers({
    @Layer("CONSOLE_UI may refer to edu.afit.smallworld.{MODEL,CONTROLLER}|IO"),
    @Layer("SWING_UI may refer to edu.afit.smallworld.{MODEL,CONTROLLER}|SWING")
})
@Promise("@InLayer(CONSOLE_UI, SWING_UI)")
package edu.afit.smallworld.textui;
```

package-info.java within the edu.afit.smallworld.textui.parser package:

```
@Promise("@InLayer(edu.afit.smallworld.textui.{CONSOLE_UI, SWING_UI})")
package edu.afit.smallworld.textui.parser;
```

package-info.java within the edu.afit.smallworld.textui.console package:

```
@Promise("@InLayer(edu.afit.smallworld.textui.CONSOLE_UI)")
package edu.afit.smallworld.textui.console;
```

package-info.java within the edu.afit.smallworld.textui.graphical package:

```
@Promise("@InLayer(edu.afit.smallworld.textui.SWING_UI)")
package edu.afit.smallworld.textui.graphical;
```

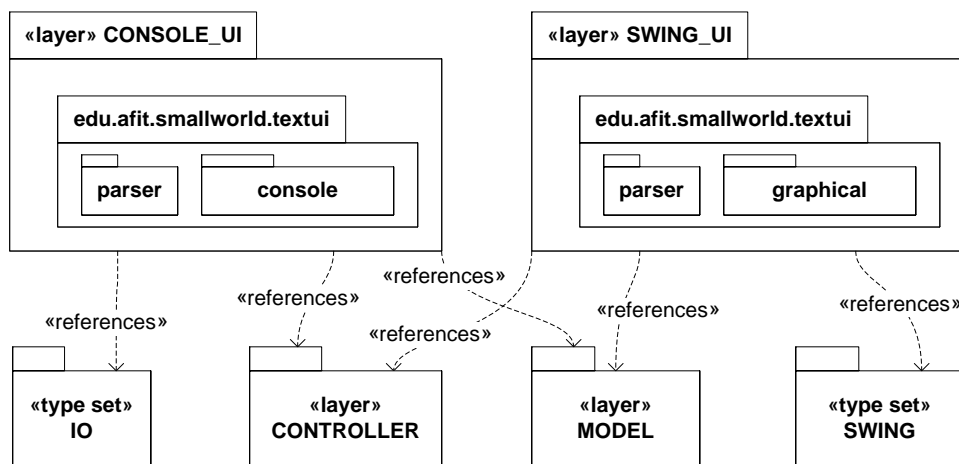


Figure 6.3: A model of the static structure of the two SmallWorld user interfaces extending the model in Figure 6.2. The promises added to the SmallWorld code are shown above a UML diagram illustrating their semantics. The first user interface interface, which allows the user to interact with the game by typing text commands within a console window, is mapped into the CONSOLE_UI layer. The second user interface, which allows the user to interact with the game by typing text commands within an AWT/SWING window, is mapped into the SWING_UI layer. Two of the Java packages, edu.afit.smallworld.textui and edu.afit.smallworld.textui.parser, are mapped into both layers. These two packages are only allowed to reference the MODEL and the CONTROLLER—the intersection of the allowed to reference sets defined for the CONSOLE_UI and SWING_UI layers.

6.2.4 Checking that layered models are well-formed

A well-formed specification of layers implies that there exists an ordering such that lower layers may not depend on higher layers. Checking this property is complicated because we allow a particular type to be mapped into more than one layer, *i.e.*, we do not require layers to define a partition of the system. Note that here we are checking if a particular layered model is well-formed, not verifying that it is consistent with code.

Definitions: We define T as the set of all types defined in the program and L as the set of all layers defined in the program.

The set of types mapped into a layer is given by

$$def : L \rightarrow \wp(T).$$

For a given layer, l , the elements of $def(l)$ can always be enumerated because types are explicitly mapped into l using the `@InLayer` annotation and `@InLayer` may only map a type into a layer if it is located within the package tree where that layer is declared.

The set of types allowed to be referenced by a layer is given by

$$ref : L \rightarrow \wp(T).$$

For a given layer, l , This set includes all the types defined by the `may refer to` clause of the layer definition of l as well as $def(l)$. Thus it is always true that $def(l) \subseteq ref(l)$ which is consistent with our intuition that types within a layer should be allowed to reference each other. Our approach avoids having to enumerate the elements of $def(l)$ because it could potentially contain every class on the program's classpath (*e.g.*, `@TypeSet("CP = Object | !(Object)")`). However, it is tractable to test if a particular type is an element of this set.

Properties: We say that a layered model is *well-formed* if

1. *Layers are ordered:* The transitive closure of the references relationship between layers must define a partial order on the set of layers defined for the system². Here we only consider the relationship between layers that can be determined by examining layer definition promises—ignoring the types mapped into them and any type sets they are allowed to reference. We use the symbol \succsim to refer to the transitive closure of the references relationship between two layers. If $l_1 \in L$ and $l_2 \in L$ then $l_1 \succsim l_2$ means that l_1 is a higher layer than l_2 . (We say that l_1 is a higher layer than l_2 if l_1 may refer to types in l_2 .) We refer to this partial order as $\langle L, \succsim \rangle$. In general, $\langle L, \succsim \rangle$ will not be a chain (*i.e.*, a totally ordered set), therefore, we use the symbol \parallel to indicate non-comparability. We write $l_1 \parallel l_2$ if $l_1 \not\succsim l_2$ and $l_2 \not\succsim l_1$.
2. *Types referenced by layers are ordered:* If the set of types referenced by a layer, l_1 , intersects with the set of types mapped into a second layer, l_2 , then l_1 must be above l_2 or be non-comparable with l_2 . More formally, for all $l_1 \in L$ and $l_2 \in L$,

$$ref(l_1) \cap def(l_2) \neq \emptyset \rightarrow (l_1 \succsim l_2 \vee l_1 \parallel l_2)$$

must hold. This constraint is necessary because we allow types to be mapped into more than one layer and we allow layers to reference any set of layers and type sets.

²A layer is always allowed to reference itself, hence reflexivity does hold.

We now return to the intuition of a layers introduced at the beginning of this section. A well-formed specification of layers implies that there exists an ordering such that lower layers may not depend on higher layers. If a layered model is well-formed, in the sense that the two properties defined above hold, is our intuitive notion of a layered system met? The existence of an order is immediate by our first property, *layers are ordered*, however, the requirement that “lower layers may not depend upon higher layers” is less obvious and is proved in Theorem 6.2.1.

Theorem 6.2.1 (Layers are ordered). *Given a well-formed layered model (i.e., where the two properties defined above hold), no type in a lower layer is allowed to reference a type in a higher layer. More formally, for all distinct $l_1 \in L$ and $l_2 \in L$ such that $l_1 \succcurlyeq l_2$, $ref(l_2) \cap def(l_1) = \emptyset$.*

Proof. (By contradiction) Assume to the contrary that there exists a type t such that $t \in ref(l_2) \cap def(l_1)$. By the *types referenced by layers are ordered* property of a well-formed layered model we know that $l_2 \succcurlyeq l_1$ or $l_2 \parallel l_1$ because $ref(l_2) \cap def(l_1) \neq \emptyset$ (i.e., it contains t). We now proceed by considering these two cases:

(Case $l_2 \succcurlyeq l_1$) If $l_2 \succcurlyeq l_1$, we know by the hypothesis that $l_1 \succcurlyeq l_2$ and that $l_1 \neq l_2$. However, this is a contradiction because it violates the antisymmetry property of the partial order $\langle L, \succcurlyeq \rangle$ specified by the *layers are ordered* property of a well-formed layered model.

(Case $l_2 \parallel l_1$) If $l_2 \parallel l_1$, we know by the hypothesis that $l_1 \succcurlyeq l_2$. However, by the definition of non-comparability this is a contradiction.

In both cases we reach a contradiction so the proposition is true. □

Example: Is the SmallWorld layered model defined in Figure 6.2 and Figure 6.3 well-formed? We check if the two properties defined above hold for this model. The set of layers is

{CONSOLE_UI, CONTROLLER, MODEL, PERSISTENCE, SWING_UI}

and the references relation between layers is

{ (CONSOLE_UI, CONTROLLER)
 (CONSOLE_UI, MODEL)
 (SWING_UI, CONTROLLER)
 (SWING_UI, MODEL)
 (CONTROLLER, PERSISTENCE)
 (CONTROLLER, MODEL)
 (PERSISTENCE, MODEL) }.

The transitive closure of this set is the partial order shown by the Hasse diagram in Figure 6.4. Hence, the *layers are ordered* property of a well-formed layered model holds for SmallWorld.

To determine if the *types referenced by layers are ordered* property holds for SmallWorld we need to find any cases where the set of referenced types in a layer overlaps with the set of types which define the contents of another layer. This occurs in only two cases: CONSOLE_UI and SWING_UI. In Figure 6.3 it can be seen that both these layers contain (and thus are allowed to reference) the contents of the `textui` and `parser` packages. The CONSOLE_UI layer references the `textui` and `parser` packages which are mapped into the SWING_UI layer.

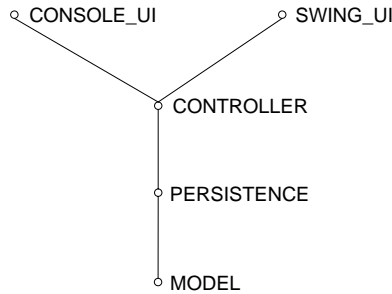


Figure 6.4: A Hasse diagram [34] of the “is a higher layer than” partial order on the set of SmallWorld layers.

The `SWING_UI` layer references the `textui` and `parser` packages which are mapped into the `CONSOLE_UI` layer. In both of these cases the *types referenced by layers are ordered* property holds because, by Figure 6.4, these two layers are non-comparable with the “is a higher layer than” partial order relation on the set of SmallWorld layers.

6.2.5 Restricting allowed references

In some cases it is clearer to specify the set of types that are allowed to reference a particular declaration rather than, as we have been doing so far, specifying that each type is allowed to reference it. One example of this is when one type manages creation of a second type, *e.g.*, within a composite structure. In SmallWorld a world is composed of its places. Thus, the constructor for the `Place` class is only intended to be called from the factory method within the `World` class. This example is modeled in Figure 6.5.

The `@AllowsReferencesFrom` promise specifies that the declaration where it is placed may only be used within the given type set. This promise may target types, fields, methods, or constructors. Thus it can be used to express lower-level design intent than `@MayReferTo` which cannot target fields, methods, or constructors. The argument to the `@AllowsReferencesFrom` promise specifies a type set. To be precise about its semantics, the `@AllowsReferencesFrom` promise specifies the set of types where the element is *allowed to be referenced*. Actual references to the targeted unit by types within the specified type set are not required to exist. `@AllowsReferencesFrom` promises must be consistent with `@MayReferTo` promises if both exist or the model is nonsensical, *e.g.*, in SmallWorld the world should be permitted to reference its places.

Example: The Eclipse project uses a programming convention which can be assured by the `@AllowsReferencesFrom` promise. The source code for Eclipse contains several packages called `internal`. The types contained within these packages are, by convention, only intended to be referenced within `org.eclipse` source code—not by outside code. This prohibition can be modeled with a `@AllowsReferencesFrom` promise as shown in Figure 6.6.

Place.java within the edu.ait.smallworld.model package:

```
public class Place {  
  
    /**  
     * Constructs a new place. Only to be invoked by the {@link World} class.  
     */  
    @AllowsReferencesFrom("World")  
    Place(World world, String name, String article, String description) { ... }  
    ...  
}
```

Figure 6.5: The constructor for Place objects is only allowed to be invoked from within the World class.

package-info.java within the org.eclipse.jdt.internal package:

```
@Promise("@AllowsReferencesFrom(org.eclipse+)")  
package org.eclipse.jdt.internal;
```

Figure 6.6: Specifying that it is prohibited for code outside the org.eclipse package tree from using types defined within the org.eclipse.jdt.internal package.

6.3 Verification

Verification of our specification of static program structure consists of two parts: (1) checking that the models are well-formed, and (2) checking that the code is consistent with the well-formed models.

6.3.1 Checking for well-formed models

After checking the syntax of each promise is valid, two properties must hold for a layered model to be well-formed: layers are ordered and types referenced by layers must be ordered.

Layers are ordered

Checking that *layers are ordered* is performed by building a directed graph of the “is a higher layer than” relationships defined between defined layers. If the graph is determined to have a cycle then the property does not hold and the problem is reported to the tool user.

Example: An example of this check on a model that defines a cycle is shown in Figure 6.7.

Types referenced by layers are ordered

Checking that *types referenced by layers are ordered* also uses the directed graph of the “is a higher layer than” relationship between layers. The graph is used to determine the set of layers that is above a particular layer being checked. The algorithm used to check this property is given in Figure 6.8. This algorithm depends upon the fact that the set of types

package-info.java within the edu.example package:

```
@Layers( {
    @Layer("TOP    may refer to BOTTOM"),
    @Layer("BOTTOM may refer to TOP")
})
package edu.example;
```

Description	Resource	Line
Ⓢ Cycle detected on Layer BOTTOM may refer to TOP	/edu.example.cycle/src/edu/example/package-info.java	1
Ⓢ Cycle detected on Layer TOP may refer to BOTTOM	/edu.example.cycle/src/edu/example/package-info.java	1

Figure 6.7: An example of a structural model that is not well-formed because the defined layers are not ordered. (Top) The definitions in the edu.example package. (Bottom) The tool view showing the report to the user in the *Modeling Problems* view about the cycle detected between the layers TOP and BOTTOM.

```
For each layer,  $l$ :
    Determine the set of layers "above"  $l$ , which we refer to as  $A$ .
    For each layer,  $l_a$ , in  $A$ :
        For each type  $t_a$  in  $l_a$  via an @InLayer promise:
            If  $t_a$  is in the set of types that  $l$  is allowed
            to reference then the model is not well-
            formed
    If no violations are found then the types referenced by layers are ordered
```

Figure 6.8: The algorithm used to check if types referenced by layers are ordered.

that a layer is allowed to reference includes the types defined to be in the layer. This fact allows the algorithm to simply ensure that each type in a higher layer is not allowed to be referenced by a lower layer.

To illustrate a violation, consider a system containing the types T1, T2, and T3 in the package com.example with the structural annotations shown in Figure 6.9. In this example the *layers are ordered* property of a well-formed layered model holds because TOP is a higher layer than BOTTOM and that is the only relationship that exists.

This model is not well-formed by the algorithm above because layer BOTTOM is allowed to reference type T3 which is part of the higher layer TOP. From the tool user's point of view, there is one of two possible modeling problems: (1) The type T3 should not be in the layer TOP, or (2) The layer BOTTOM should not be allowed to reference the type T3. The tool reports this problem in the manner of the second. In addition, as shown in Figure 6.9 we report this particular modeling problem in the *Verification Status* view as an inconsistency. Why take this odd approach? The reason is that we found it very difficult to communicate a problem

of this complexity to the tool user in the *Modeling Problems* view. A single line explanation was not descriptive enough for the user to understand and take corrective action. In addition, because we take the point of view that the lower layer is “broken” we can report verification results to the programmer about higher layers (*e.g.*, TOP in Figure 6.9).

6.3.2 Checking model–code consistency

The verification that a well-formed model of static structure is consistent with the code proceeds as follows. For each compilation unit, *c*, with a structural assertion we simply check that every reference to other types within *c* is allowed by the structural assertion.

This analysis is not flow-sensitive and is performed by binding each use in a compilation unit to its definition (*i.e.*, if the definition is not itself a type) is then checked to ensure that it is allowed to be used.

Figure 6.10 shows the verification results for the structural model about the SmallWorld program shown in Figure 6.2. The model is not fully consistent with the code. The MODEL and PERSISTENCE layers are consistent with the code, however the CONTROLLER layer is not. If a compilation is consistent then only a single green “+” is reported for the entire compilation unit. This is shown for the `Direction` compilation unit which is in the MODEL layer. If a compilation unit is inconsistent then a red “×” is reported for each use of a type the compilation unit is not supposed to depend upon. This is shown by the 5 red “×” results for the `WorldController` compilation unit. This compilation unit is referring to several types within `java.util`. Each result links the tool user to the exact line of code where prohibited reference is made. In this case, however, it seems reasonable to allow the CONTROLLER layer to reference types within `java.util`. In fact, we had defined a type set for this purpose:

```
@TypeSet("UTIL=java.util & !(java.util.{Enumeration, Hashtable, Vector})")
```

By changing the definition of the CONTROLLER layer to

```
@Layer("CONTROLLER may refer to MODEL | PERSISTENCE | UTIL | java.io.File")
```

the JSure tool can now verify the defined model of static structure as shown in Figure 6.11.

package-info.java within the edu.example package:

```
@Layers({
    @Layer("TOP    may refer to BOTTOM"),
    @Layer("BOTTOM may refer to edu.example.T3")
})
package edu.example;
```

T1.java within the edu.example package:

```
@InLayer("TOP")
public class T1 { ... }
```

T2.java within the edu.example package:

```
@InLayer("BOTTOM")
public class T2 { ... }
```

T3.java within the edu.example package:

```
@InLayer("TOP")
public class T3 { ... }
```

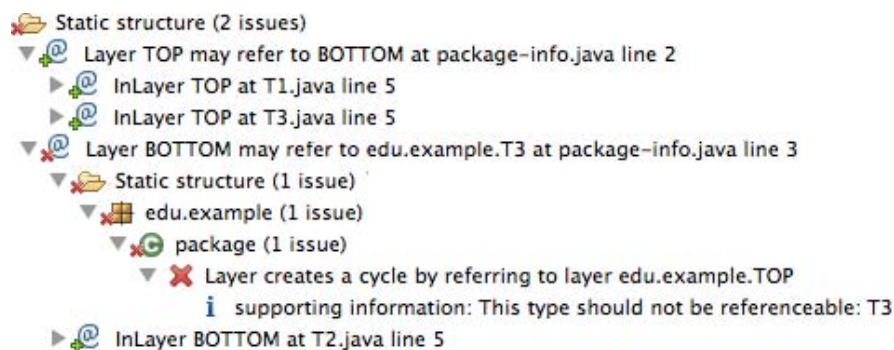


Figure 6.9: An example of a structural model that is not well-formed because the types referenced by layers are not ordered. (Top) A specification of static structure of the layers `TOP` and `BOTTOM` in the package `edu.example` and three types within that package. (Bottom) The tool view reporting that the `T3` type should not be referenceable by the `BOTTOM` layer.

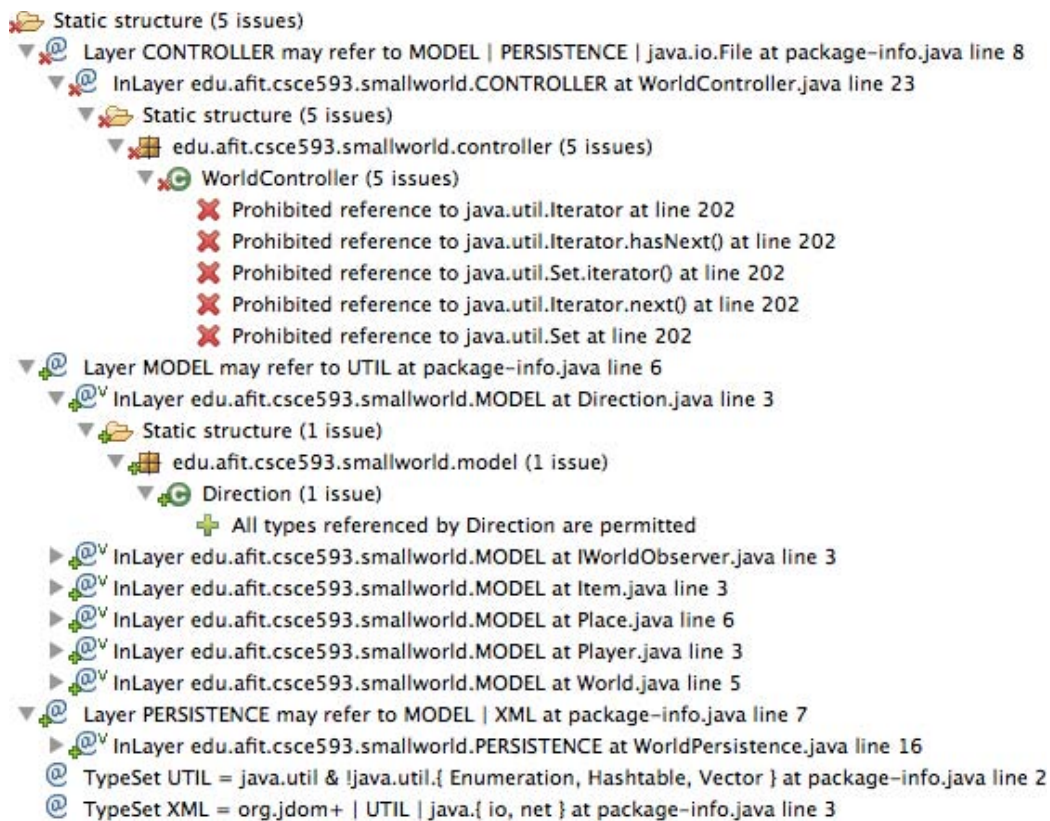


Figure 6.10: Tool reported verification results for the structural model about the SmallWorld program shown in Figure 6.2.

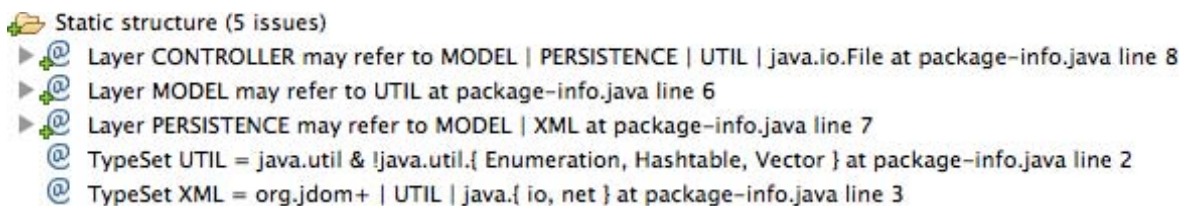


Figure 6.11: Tool reported verification results for the structural model about the SmallWorld program shown in Figure 6.2 when the CONTROLLER layer is changed to allow references to the typeset UTIL.

6.4 Not a module system

Our approach to the specification and verification of static program structure is not what is commonly referred to as a *module system*. Both deal with constraining the static structure of a program. Our approach is primarily focused on, for a particular type, defining what other types that type is allowed to reference. A module system is primarily focused on defining and enforcing an interface, in terms of types and methods, that a collection of types presents to the rest of the system. Roughly speaking, our approach answers the question for a component, “what am I allowed to use?” While a module system answers the question, “What parts of me are others allowed to use?”

Several improved module systems have been proposed for the Java programming language. These include Jiazzi [80], aimed at supporting mix-ins, MJ [31], which adds static modules to Java, and the Fluid Module System [103], which is similar to MJ but is hierarchical. A standardization effort, through Java Specification Request (JSR) 294: Improved Modularity Support in the Java Programming Language³, is ongoing.

It would be desirable for our approach to be able to reference the modules defined by these systems in layers. For example, consider the following partial specification of a module in the Fluid Module System.

```
@Module("Swing")
public class ...
```

This annotation on the desired classes coupled with visibility annotations defines a module named *Swing*. It seems reasonable to allow layer definitions to refer to this named module.

```
@Layer("UI may refer to Swing | java.util")
```

We leave the detailed design and implementation of this feature as future work. We note, however, that our current approach is only compatible with systems focused on static module definition, approaches that elicit the composition of and connections between components at runtime, such as ArchJava [3], are not as straightforward to support.

6.5 Conclusion

In this chapter we have presented a new approach to the specification and verification of the static structure of Java programs. Our approach builds upon the demonstrated utility to practicing programmers of *reflexion models* by Murphy and Notkin [85]. Our approach combines the creation of the high-level model and a mapping from the high-level model to the source code via source code annotations to allow programmers to express and maintain the static structure of their code. Our primary contribution to prior work is the addition of a lightweight approach to specify and verify static layers with well defined semantics that we believe are consistent with traditional layered semantics. In addition, our approach more naturally facilitates composition of multiple overlapping static models.

This work is presented, in the context of this thesis, as evidence that analysis-based verification is scalable with respect to new assurance attributes, in particular non-concurrency related program attributes.

³<http://jcp.org/en/jsr/detail?id=294>

Validation

“The logic of validation allows us to move between the two limits of dogmatism and skepticism.” — Paul Ricoeur

In this chapter we recapitulate the evidence we provide in support of this research and present a cost-effectiveness analysis of analysis-based verification informed by our trials of a prototype tool in the field. Section 7.1 summarizes the evidence presented throughout this dissertation in support of the claims of this research. Section 7.2 provides rough estimates for the cost and benefit a potential tool user may incur.

7.1 Sound combined analyses

The key idea and overall vision of the Fluid research group is focused *analysis-based verification* for software quality attributes as a scalable and adoptable approach to the verification of consistency of code with its design intent. Our principal contribution to this vision is the development of the concept of *sound combined analyses* for the verification of mechanical program properties. This includes (1) meta-theory, (2) user experience design and tool engineering approach, and (3) field validation. Figure 1.22 (repeated from Chapter 1) illustrates the relationship among the vision of the project, the three principal contributions of this thesis, and the enabling sound analysis work of Greenhouse, Boyland, and Sutherland. The field validation is used to validate the overall vision of the Fluid project as well as our contribution. It has also informed the development of our work.

We now summarize the evidence presented for each principal contribution of our work.

(1) **Meta-theory** to establish soundness of the approach of combining multiple constituent sound static analyses (*e.g.*, binding context, effects upper bounds, uniqueness) into an aggregate developer-focused analysis (*e.g.*, safe lock use).

The meta-theory developed in this thesis is presented in Chapter 2. This chapter demonstrates that our verification proof calculus supports the construction of verification proofs from fragmentary analysis results reported by multiple underlying constituent analyses. We establish soundness of our verification proof calculus by a proof (of Theorem 2.7.4) that re-

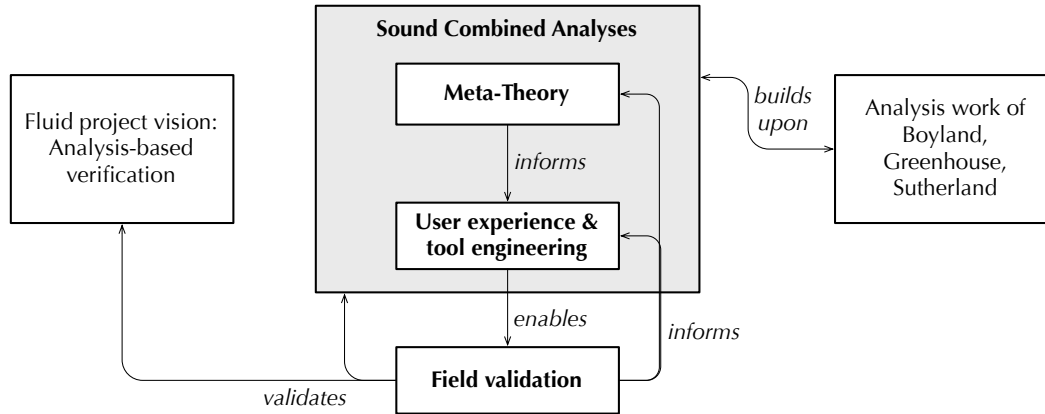


Figure 7.1: A sketch of how the principal contributions of this thesis (in bold at the center) relate to the overall vision and prior work of the Fluid project at Carnegie Mellon University.

lates a semantics of fragmentary analysis results to broad conclusions regarding the program being analyzed. Several key lemmas (Section 2.6 and Section 2.7) are proved in support of the soundness theorem about the verification proof calculus and the precise semantics of fragmentary analysis results. Our proofs assume the soundness of the underlying constituent analyses.

We suggest that our assumption that constituent analyses used in the prototype tool (as listed in Figure 1.3) are sound is reasonable based upon prior work [21, 20, 53, 103]. Prior work has not demonstrated each analysis is sound via rigorous proof. The approach taken is to provide a precise definition of static semantics for each analysis. The static semantics are presented through an extension of Featherweight Java [65]. Featherweight Java bears a similar relation to Java as the lambda-calculus does to languages such as ML and Haskell. Analysis implementation was informed by this precise model and tested on a large corpus of Java code to demonstrate that it produces conservative results.

(2) **User experience** design and **tool engineering** approach designed to address adoption and usability criteria of professional development teams.

The JSure prototype tool, incorporating the prior work of the Fluid project on sound analysis, was developed as part of our work. The tool provides evidence that our approach, with its constituent analyses, is *feasible* for the tool-supported verification of non-trivial narrowly-focused mechanical properties about programs with respect to explicit models of design intent. The engineering of this tool (and its user experience) is introduced in Chapter 1 and presented in further depth in Chapter 3.

The design intent language (*i.e.*, promises) and verifying analyses are predominantly prior work by the Fluid research group (as summarized in Figure 1.3 and presented in Section 1.9). However, all of this analysis work fundamentally depends upon our approach to be understandable by practicing programmers. All of these analyses use the drop-sea proof management system (Section 3.4) to report and manage their results. These constituent analyses use our approach to produce programmer-meaningful verification results—without the contributions of our work they produce only large numbers of unorganized fragmentary results about the code they examine.

Our engineering work provides a technical and software framework, realized in the JSure tool, for the verification of mechanical program properties. Two case studies of adding new aggregate analyses to the JSure tool are used to qualitatively evaluate the benefits, as well as identify limitations, of this verification framework (*e.g.*, drop-sea, scoped promises, proposed promises, the red dot), with respect to *scalability* when new attributes are added. These case studies (*thread coloring* and *static layers*) are presented in Chapter 5. The case studies show that our work helps to facilitate the addition of new sound analyses into the JSure prototype tool by providing a verification framework that simplifies implementation and by providing capabilities, such as @Promise, that are widely needed. Figure 5.1 summarizes the perceived benefits and limitations of capabilities provided by our verification framework from the point of view of an analysis author. The limitations identified are left to be addressed in future work.

(3) **Field validation** in collaboration with professional engineers on diverse commercial and open-source code bases.

Nine trials using the prototype JSure tool in the field on open source, commercial, and government Java systems are presented in Chapter 4. As illustrated in Figure 7.1, our field trials provide empirical evidence in support of the contribution of our work as well as the overall vision of the Fluid project. The evidence from the field trials supports claims about use of the JSure prototype tool by disinterested practitioners on code that they develop and maintain. These claims, with a summary of the supporting evidence (Section 4.3), are:

- *The JSure prototype tool scales up to use on large real-world software systems.* The empirical evidence is the successful use of the tool on 20 client software systems examined over 25 days during nine field trials. The largest client system examined was 350 KSLOC and the mean size was 90 KSLOC.
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that the tool was effective with respect to defects found.* The quantitative evidence with respect to defects found is that across the nine field trials the JSure tool helped to identify 79 race conditions in 1.6 million lines of real-world Java code—most of which had already passed organizational acceptance evaluation for deployment. The client developers described the difficulty of finding these defects, “It would have been difficult if not impossible to find these issues without [JSure].” and “[JSure] identified logic and programming errors . . . that extensive review and testing did not discover.”
- *At the conclusion of a 3-day field trial using the JSure prototype tool, the behavior and responses of the client developers indicate that they perceive value from the verification results obtained.* The quantitative evidence with respect to verification results is that across the nine field trials we developed 376 models of programmer intent about lock use and were able to verify most of them with the tool by working alongside client programmers. The client developers described the value of the verification results, in particular the value of the tool’s specification language, “[JSure] was reported by all participants as helping them to understand and document the thread interactions that they had already designed and implemented.”
- *Within two or three hours of using the JSure prototype tool to annotate and analyze the client’s code, the behavior and responses of the client developers indicate that they*

perceive sufficient reward to continue use. The behavioral evidence is continuing to participate in the engagement and inviting other developers to participate. This was observed in 8 of the 9 field trials. The client developers described the value of the immediacy of results, “We found a number of significant issues with just a few hours of work. We really like the iterative approach.”

- *It is feasible to adopt the JSure prototype tool late in the software engineering lifecycle.* The empirical evidence is that 18 of the 20 client software systems examined in the field were in the operations and maintenance phase of the software lifecycle—they had already passed organizational acceptance evaluation for deployment. One of the commercial J2EE servers examined had been in release for 3 years. The verbal evidence consisted of expression by client developers advocating widespread use of the tool throughout the software lifecycle (when code exists), such as, “I can’t think of any of our Java code I wouldn’t want to run this tool on.”

Generally speaking, the field trials show that analysis-based verification is valued by disinterested practitioners—working programmers whose only interest in our work is the immediate value that JSure can potentially provide to them on code they develop and maintain.

7.2 Cost-effectiveness analysis

Our experience with analysis-based verification on production open source, commercial, and government Java systems enables us to provide rough estimates for the cost and benefit a potential user may incur. In this section we consider our approach, sound combined analyses, coupled with the prior sound analysis work of Greenhouse, Boyland, and Sutherland as realized in the JSure analysis-based verification tool.

For the purposes of this analysis, we define cost to be developer effort, both initially in an engagement and incrementally in adding model information and undertaking analysis effort.

7.2.1 Cost

Use of JSure involves programmer time and effort to express models of design intent, typically as annotations in the code. Features introduced by this thesis, *e.g.*, proposed promises and scoped promises, provide tool assistance with model expression—helping to lower this cost. Preliminary work using the *Flashlight* dynamic analysis tool to perform model inference (as summarized in Figure 4.13) may also help to lower this cost.

A user of JSure must also invest time and effort to learn how to use the tool, its annotation language, and how to interpret tool results. The field trials demonstrate that professional development teams are able to learn to use the tool and obtain results useful to them within two or three hours of initial contact with the technology and the tool (Section 4.3).

7.2.2 Benefit

A user of JSure gets the benefit of automated verification of expressed models of design intent. In the absence of any user-introduced red dots, each verified annotation is an invariant of the

program, *i.e.*, the assertion made by the annotation holds for all possible executions of the program.

In addition to automated verification, the process of expressing models of design intent and using the tool provides two secondary benefits: (1) precise documentation and (2) defect finding.

- (1) **Precise documentation:** The use of the JSure annotation language precisely and tersely documents programmer design intent (*e.g.*, intended lock use, intended thread use). Client programmers who participated in the field trials of the JSure tool placed great value on the documentation of design intent provided by the tool’s annotation language (Section 4.3).
- (2) **Defect finding:** Through interaction with the tool a user of JSure gets the benefit of finding defects in the code. In the area of lock use, the field trials uncovered 79 race conditions in 1.6 million lines of production Java code—most of which had already passed state-of-the-practice organizational acceptance evaluation for deployment (Section 4.3). All of these code defects were uncovered by the tool identifying code that was inconsistent with the models of lock use intent expressed by the programmers.

7.2.3 Alternative approaches

In this section we consider the cost-effectiveness of analysis-based verification, as realized in the JSure prototype tool, relative to other alternative approaches. We focus on the concurrency-related program properties verifiable by the tool.

Alternative approaches include primarily inspection and testing. For the kinds of errors addressed by JSure, and particularly concurrency errors, these approaches can be expensive, inaccurate, or limiting on functionality or performance. This was validated in two ways: First, through interactions with the professional developers, who had employed the best practice approaches for the critical systems we were shown, and had nonetheless not been able to “catch” the defects identified through the use of the JSure tool:

“Held successful Fluid workshop on software for the U.S. Navy under development at ESBA MS2 Moorestown 19–21 July. Team developed 63 lock models and [JSure] identified logic and programming errors in the Common Sensor and Tracking (CSAT) services and Weapons Control Engagement segments that extensive review and testing did not discover.” [72]

Second, through analysis of the literature related to our work.

The alternative approaches faced by developers are (1) testing, (2) inspection, (3) modeling checking, (4) heuristic-based static analysis, (5) avoid concurrency.

- (1) **Testing:** The professional developers the we worked with told us that testing for concurrency involves long running integration “stress” test on a variety of concurrent hardware. The longer the system is able to run without crashing the more confident the developers are that the implementation is correct with respect to concurrency. Use of the JSure tool in the field still uncovered concurrency defects in these systems. Our

results in the field indicate that testing is ineffective in eliminating concurrency defects from code (Section 4.3).

- (2) **Inspection:** Several of the organizations we worked with in the field use inspection to uncover defects and improve code quality. Similar to testing, our results in the field indicate that inspection is ineffective in eliminating concurrency defects from code (Section 4.3). Inspection is time consuming for programmers. Programmers noted during the field trials that a tool like JSure would help to focus code inspections and save them considerable time:

“... Another engineer observed, [JSure] ‘pointed out things that we would not have looked at—would not have even noticed in a code review.’ ” [25]

- (3) **Model checking:** Software model checking is an area of active research and holds promise to find widespread use in practice. One successful example is the SLAM tool at Microsoft, which targets consistency of device driver code with protocol requirements associated with the Windows device driver API [8]. SLAM is limited with respect to the scale of the code bases that it can examine (~10 KSLOC). Several model checking approaches, for example JavaPathFinder¹, require the disciplined use of a subset of the programmer language. This limitation makes them costly to adopt at scale and late in the software lifecycle.
- (4) **Heuristic-based static analysis:** Heuristics-based static analysis tools, such as Find-Bugs, have been successful in uncovering a large number of defects in real-world code (Section 1.3.2). There are several costs. (1) The cost of not being sound: missed defects. (2) The cost in programmer time and effort of dealing with large numbers of false positives and findings of marginal significance to current developer concerns.
- (5) **Avoid concurrency:** This approach may be feasible for some systems, however, due to the current trend of hardware becoming more aggressively parallel at constant (or diminishing) clock speeds [51] such systems are likely to sacrifice performance and responsiveness. For other systems this approach is not feasible. It is not possible to build a modern application server, such as the two J2EE servers that the research team examined in the field, that isn’t concurrent. It is also not possible to build a MapReduce infrastructure, such as Hadoop, that isn’t concurrent.

The cost of the alternative approaches can be reduction in functionality (due to the need to simplify structure and concurrency) or increased hazard in operations (due to the fact that races and other issues went unresolved). At this point in development, we are not aware of alternatives that are both sound and scalable and that can provide analysis-based verification for concurrency.

¹<http://javapathfinder.sourceforge.net/>

Conclusion

“Let’s not fall prey to the syndrome of accepting a wish, stated with a fancy name, as an established capability.” — Michael L. Dertouzos

Dertouzos was referring to intelligent agents, which he argued (in 2001) were not all that intelligent. We suggest that his comment also applies to practical verification systems. Indeed, we echo his further thought that the problem is central and is perhaps receiving insufficient attention today as a consequence of a record “of past disappointments.” [38] Our work contributes to the quality assurance of software by providing a program verification approach that, because it is focused on narrow software quality attributes, is scalable and composable. The approach is based on a novel technique (elaborated in Chapter 2) for combining multiple constituent sound analyses into an overall verification technique. We therefore refer to the overall approach as *sound combined analyses* for *analysis-based verification*.

8.1 Summary of contributions

The key idea and overall vision of the Fluid research group is focused *analysis-based verification* for software quality attributes as a scalable and adoptable approach to the verification of consistency of code with its design intent. Our principal contribution to this vision is the development of the concept of *sound combined analyses* for the verification of mechanical program properties. These include

- **Meta-theory** to establish soundness of the approach of combining multiple constituent sound static analyses (*e.g.*, binding context, effects upper bounds, uniqueness) into an aggregate developer-focused analysis (*e.g.*, safe lock use) (Chapter 2).
- **User experience** design and **tool engineering** approach designed to address adoption and usability criteria of professional development teams (Chapter 3).
- **Field validation** in collaboration with professional engineers on diverse commercial and open-source code bases (Chapter 4).

Our work, incorporating the prior sound analysis work of the Fluid project, led to the development of the JSure prototype analysis-based verification tool. This tool was used in

our field validation. The field validation presented in Chapter 4 validates the overall vision of the Fluid project as well as our contribution. It has also informed the development of our work.

Other new technical and engineering results contributed by this thesis include:

- **The drop-sea proof management system:** Drop-sea is the proof management system used by the JSure tool. Drop-sea manages the results reported by constituent program analyses and automates the proof calculus presented in Chapter 2 to create verification results based upon these findings. (Chapter 3)
- **Management of contingencies—the red dot:** Drop-sea allows several unverified contingencies to exist in a chain of evidence about a promise. A programmer can vouch for an overly conservative analysis result—changing it from an “x” to a “+”. A programmer can turn off a particular program analysis causing all the promises checked by that analysis to have no results—causing the tool to trust these promises without any analysis evidence. Finally, the programmer can assume something about a component that is outside of the programmer’s scope of interest (*e.g.*, on the other side of an organizational or contractual boundary). These actions introduce a contingency into any proof that relies upon them. Drop-sea explicitly tracks these contingencies and flags them with a red dot. (Chapter 3)
- **Proposed promises:** Our approach has constituent analyses report any necessary prerequisite assertions as part of each analysis result. Analyses, when they report a prerequisite assertion, propose promises that may or may not exist in the code. A special analysis called *promise matching* is used to “match” each proposed promise with a programmer-expressed promise in the code. If no “match” can be found, *i.e.*, a promise proposed by a constituent analysis is not in the code base, then the computation that produces verification results is able to use the unmatched proposed promises to determine the “weakest” prerequisite assertion for each promise in the code base. This allows the tool to propose “missing” annotations, from the point of view of the constituent analyses, to the code that can be reviewed and accepted by the tool user. (Chapter 3)
- **Scoped promises:** Scoped promises are promises that act on other promises or analysis results within a static scope of code. We introduce three types of scoped promises: `@Promise` to avoid repetitive user annotation of the same promise over and over again in a class or package, `@Assume` to support team modeling in large systems where programmers are not permitted access to the entire system’s code, and `@Vouch` to quiet overly conservative analysis results. Scoped promises help to “scale up” the ability of a programmer or a team of programmers to express design intent about a large software system. (Chapter 3)
- **An approach to the specification and verification of static program structure:** Previous work by Murphy and Notkin [85] has demonstrated the utility to practicing programmers of tool support to understand and maintain structural models of their code. Our approach combines the creation of the high-level model and a mapping from the high-level model to the source code via source code annotations—primarily a syntactical difference—but our purpose is the same: to help programmers express, understand, and maintain the static structure of their code. Our primary contribution to prior work is the addition of a lightweight approach to specify and verify static

layers with well defined semantics that we suggest are consistent with traditional layered semantics. In addition, our approach more naturally facilitates composition of multiple overlapping static models. (Chapter 6)

8.2 Looking forward

In previous chapters we have proposed several areas for future work. These include merging our approach to the specification and verification of static program structure with Sutherland’s Fluid module system [103], improving the overall user experience of our prototype tool, adding a selective verification capability to our IDE-based tool to improve interactive performance, and the model inference capability provided by the **Flashlight** dynamic analysis tool. In this section we speculate on several other possible future directions for this work.

8.2.1 Accommodating negative analysis results

While it is undecidable in general, due to Rice’s theorem [95], there can be cases where you can definitively conclude that a promise does not hold. So in many respects, a monotonic three-valued logic, *i.e.*, **U**, **T**, and **F** (representing *unknown*, *consistent*, and *inconsistent*, respectively) with $U < T$ and $U < F$, would be a better representation for the foundations of analysis-based verification than the two-valued logic, *i.e.*, **U** and **T**, we used in Chapter 2. Kleene’s extensions of the conventional Boolean connectives (*i.e.*, \wedge , \vee , and \neg), which gives the strongest result consistent with monotonicity, appears to be a reasonable starting point [70].

This extension of our work, while interesting theoretically, does not yet have a compelling use case in practice to justify its development. Hybrid analysis-based assurance tools (see below) may provide one.

8.2.2 Supporting query-based modeling

Our approach uses promises, typically realized as annotations in the code, to represent programmer design intent. An alternative approach to model expression is for the programmer to perform a series of code queries and then ask to our tool to, “keep this result.” The query forms a model of programmer design intent that is persisted and verified by our tool. For example, a programmer queries what types a particular compilation unit, c , requires to compile. The tool reports that types t_1 , t_2 , and t_3 are currently necessary. The programmer saves this result as a model. Later, a second programmer adds a dependency to t_4 to c and the verification is reported to be inconsistent and the change made by the second programmer can be (1) backed out or (2) the query-based model can be updated to allow t_4 .

The applications of query-based modeling are not limited to static verification within a tool like **JSure**. The **Flashlight** tool collects large volumes of data that are queried with a user-expandable set of 50 queries. Any series of dynamic queries could also be saved as a model to be checked in subsequent dynamic analysis of the same program, *e.g.*, an instrumented run of the program could be added to an automated quality assurance build that is checked daily.

8.2.3 Hybrid analysis-based assurance tools

Our experience with JSure and Flashlight indicate that attribute focused tools based upon hybrid analysis techniques is worth pursuing in future work. Our results using dynamic analysis to propose lock use models that can be verified sound static analysis shows that this hybrid approach has the potential to save a significant amount of programmer modeling time. We leave to future work improving the ability to further combine static analysis (both heuristic and sound), dynamic analysis, automated theorem proving [96], and model checking [68].

JSure modeling guide

This appendix introduces the language used to specify design intent for the JSure analysis-based verification tool. The contents of this appendix were originally written by Greenhouse with later additions by various members of the JSure development team including the author. The specifications of programmer design intent are referred to as promises. *Promises* are supra-linguistic formal annotations to programs introduced by Chan, Boyland, and Scherlis in [26]. Each promise has a *precise meaning* and constrains the implementation and evolution of the code it targets. Promises are also (typically) *modular*, meaning that the implementation constraint on the code of a promise is limited in scope.

This appendix provides an overview, with several concrete examples, of the syntax and semantics of promises. However, it is not a complete reference.

- *Lock policy (Section A.1)*: Promises for lock-based concurrency are introduced by Greenhouse and Scherlis in [56] and further detailed in Greenhouse’s dissertation [53]. Experience using these promises on real code is summarized in [55].
- *Method effects (Section A.2)*: Promises for method effects are introduced by Greenhouse and Boyland in [54].
- *Unique fields (Section A.3)*: Promises for unaliased fields are introduced by Chan, Boyland, and Scherlis in [26].
- *Thread effects (Section A.4)*: The promises and analyses to support thread effects were developed as part of this work by the author and Greenhouse (in particular, to better express and verify design intent about `util.concurrent` as is described below).
- *Scoped Promises (Section A.5)*: Scoped promises are promises that act on other promises within a static scope of code. Scoped promises were developed as part of this work by the author.
- *Programmer vouches (Section A.6)*: Vouches are used to change any inconsistent analysis results within a scope of code to consistent results. This capability is used for documentation and to quiet overly conservative analysis results.

A.1 Lock policy

This section describes promises for lock-based concurrency.

A.1.1 A straightforward model

The common Java idiom whereby an object protects itself is declared by annotating a class with `@RegionLock`.

```
@RegionLock("Lock is this protects Instance")
public class C {
    protected int f;
    ...
}
```

This single class annotation does three things

1. Declares a new lock named `Lock`. The lock name enables consistent reference to the lock object in other annotations.
2. Identifies that lock with instances of the class (`this`)
3. Protects all the fields in instances of the class (the region `Instance`). Regions are described in more detail in Section A.1.6; for now it is enough to know
 - All classes have an `Instance` region.
 - The region contains all the non-`static` fields of the class.

In other words, all accesses to instance variables, for example `f`, of objects of class `C` must be within blocks `synchronized` on the instance.

The following two methods of class `C` would verify

```
public synchronized set(final int value) {
    this.f = value;
}

public int get() {
    synchronized (this) {
        return this.f;
    }
}
```

The following method of `C`, however, would not because although the method correctly protects the read of the field `f` of the object referenced by `other`, it does not protect the write to the field `f` of the object referenced by `this`:

```
public void copy(final C other) {
    synchronized (other) {
        this.f = other.f;
    }
}
```

To make this code verify (*i.e.*, make it consistent with the locking model) it needs to be holding two locks¹:

```
public synchronized void copy(final C other) {  
    synchronized (other) {  
        this.f = other.f;  
    }  
}
```

An aside about annotating Java 1.4 code

The ability to annotate Java declarations, as we did to the `C` class declaration with `@RegionLock` above, was introduced in version 5 of the Java programming language. While Java 5 (or higher) use is nearly ubiquitous in practice, there are still some teams that develop using Java 1.4. Java 1.4 does not support the annotation of Java declarations as shown above. JSure support annotation of Java 1.4 code by placing the annotations in Javadoc about the declaration. The annotation of the above locking model in Java 1.4 is:

```
/**  
 * @annotate RegionLock("Lock is this protects Instance")  
 */  
public class C {  
    protected int f;  
    ...  
}
```

The `@` in the Java 5 annotation is replaced with the `@annotate` Javadoc tag. The annotation must occur within a Javadoc block. Multiple annotations may be made in the same Javadoc block, however, each must begin with its own `@annotate` tag. This approach makes it simple to inform the Javadoc processor to ignore the `@annotate` tag for documentation processing (this would be far more complex if all JSure annotations were allowed as Javadoc tags).

A.1.2 Extending the model: Caller locking

A frequent exception to the basic lock model is the expectation by a method implementation that it is the responsibility of the caller of the method to acquire the lock. Analysis can verify this expectation if it is declared using the `@RequiresLock` method annotation:

```
@RegionLock("Lock is this protects Instance")  
public class C {  
    private int f;  
  
    @RequiresLock("Lock")  
    public void m() { ... }  
    ...  
}
```

¹This code is deadlock prone unless it's used in conjunction with a strict lock hierarchy.

When analyzing the implementation of method `C.m()`, analysis assumes that lock `Lock` is held. When analyzing the callsite of the method, however, analysis requires that the calling context hold the lock object identified with `Lock`. The method implementations `C.m()` and `C.calls_m()`, below, are thus both correct:

```
@RequiresLock("Lock")
public void m() {
    this.f = 0;
}

protected synchronized void calls_m() {
    this.m();
}
```

The method implementation of `Other.bad()` is, however, inconsistent with the model because it does not acquire the lock on the object referenced by `cObject`:

```
public class Other {
    ...
    public bad(final C cObject) {
        cObject.m(); // bad callsite!
    }
}
```

A.1.3 Extending the model: Aggregating arrays and other objects

An array in Java is a separate object from the object whose field refers to the array. Protecting an array-typed field thus protects *the reference to the array only*. It is not sufficient to extend the protection to the elements of the array: we also need to know that the array object is accessible through that field only. If the array could be referenced through other fields, then it would still be possible to access it concurrently because the locking model could be bypassed by accessing the array through a different field.

Much of the time, however, it is not intended that an array is aliased; in these cases, the array can be incorporated into the state of the object that references it. We call this *aggregating state*. An array is aggregated into the object that references it by adding a pair of annotations to the field that references the array:

```
@Unique
@Aggregate
private Object[] myArray;
```

This does two things:

1. Declares the programmer's intent that the field is the only field that references the array object it references. An analysis is used to verify that a `@Unique` field is never aliased. A `new` expression always creates an unaliased object, so it is always safe to assign the results of a new expression to a `@Unique` field.
2. Extends the state of the referencing object to include the elements of the array. State aggregation is not automatically transitive; thus if the array elements are objects, those objects are not aggregated, only the references to them.

In the example below, the constructor `Buffer(Object[])` does not verify because it assigns an array to field `buf` that might be aliased, conflicting with the `@Unique` annotation on the field. The assignment to `buf` in the constructor `Buffer(int)` is verifiable because it assigns a fresh array to the field. The lock `Lock` does not need to be held to access the field `buf` because it is `final`. The implementation of `copyContents()` does not verify because even though `Lock` does not need to be held to access the field `buf` or the field `length`, `Lock` must be held to access the contents of the array referenced by `buf`. Analysis knows that the method `System.arraycopy()` reads from the `Instance` region of its first parameter, and thus via aggregation, reads from the `Instance` region of the `Buffer` object. (This analysis is informed by effects analysis and annotations, described in Section A.2.)

```
@RegionLock("Lock is this protects Instance")
public class Buffer {
    @Unique
    @Aggregate
    private final Object[] buf;
    ...
    public Buffer(int size) {
        this.buf = new Object[size]; // good!
    }
    ...
    public Buffer(Object[] newBuffer) {
        this.buf = newBuffer; // bad!
    }

    public Object[] copyContents() {
        final Object[] copy = new Object[this.buf.length];
        System.arraycopy(this.buf, 0, copy, 0, this.buf.length);
        return copy;
    }
}
```

Aggregation applies not only to arrays, but to objects in general. So any object-typed field can be declared `@Unique` and can then have its state aggregated into the state of its referring object. In practice, aggregation of collections is common. The code below aggregates the set `log` into the `Instance` region.

```
public class LogExample {
    @Unique
    @Aggregate
    private final Set<String> log = new HashSet<String>();
    ...
}
```

The annotation syntax for aggregation when a named region is used is slightly different. Declaring a named region is described further in Section A.1.6. The code below declares a named region called `LogData` and aggregates the set `log` into it.

```
@Region("private LogData")
public class LogExample {
    @Unique
    @AggregateInRegion("LogData")
    private final Set<String> log = new HashSet<String>();
}
```

```

} ...

```

A.1.4 Constructors

Constructors cannot be declared `synchronized` in Java, but our analysis requires that fields protected by a lock be accessed only when that lock is held. So how do we keep verification from returning an inconsistent result when analyzing a constructor? We rely on the fact that during object construction, an object is almost always accessed by a single thread only: the thread that invoked the constructor. When this is the case, we can proceed as if the locks for the object's state are already held. To support that assumption we need to annotate each constructor.

One way that analysis can verify the thread confinement of a constructor is to leverage the tool's ability to verify that the constructor does not alias the constructed object—that is, that it does not create an alias to `this` during construction.

```

public class C {
    @Unique("return")
    public C(...) { ... }
    ...
}

```

The `@Unique("return")` annotation, which for a constructor is defined to be equivalent to a `@Borrowed("this")` annotation, is further described in Section A.3. In particular, when analysis knows that the constructor does not create such an alias, it also knows that it is impossible for another thread to obtain a reference to the object under construction during the constructor's execution.

Annotating a constructor as being `@Unique("return")` requires that the super-constructor it invokes is also `@Unique("return")`.

Field initialization and implicit constructors

Field initializers are part of the object construction process, and fields that have initializers are considered to be written to. Instance initializer blocks are also part of the object construction process and need to be verified accordingly. This can become problematic when a class does not have an explicit constructor:

```

@RegionLock("Lock is this protects Instance")
public class C {
    private int f = 1;
    private int g;
    {
        g = 2;
    }
    public synchronized int getF() {
        return f;
    }
    public synchronized void incG() {
        g += 1;
    }
}

```



```
}
}
```

The lock analysis will not verify the correct use of `Lock` because the accesses to fields `f` and `g` during construction are not protected. One way to fix this is to make the constructor explicit and annotate it:

```
@RegionLock("Lock is this protects Instance")
public class C {
    ...
    @Unique("return")
    public C() {}
    ...
}
```

Alternatively, if it is undesirable to insert the constructor explicitly, the constructor can still be annotated using a scoped promise. Scoped promises are described in more detail below, but for the case of annotating an implicit constructor, we would annotate the class as follows:

```
@RegionLock("Lock is this protects Instance")
@Promise("@Unique(return) for new()")
public class C {
    ...
}
```

A.1.5 Thread-safe objects

Region aggregation, described above, is one technique that can be used to deal with the fact that fields reference objects. But it is not always possible to use region aggregation to simplify reasoning about protected state because a field may be aliased. In such cases, the tool may produce warnings that a reference of the form `e.f.g`, where `f` is field protected by a lock, is a “possibly unsafe reference to protected shared state.” The message is meant to remind the programmer that although the field `f` is protected by a lock, this lock does not also protect the field `g` of the referenced object. (If field `f` is of class `C` and class `C` declares that `g` is protected by a lock then this warning is not produced: the tool instead attempts to verify that the appropriate lock for `g` is held.) There are situations where invoking a method via `e.f.m()` will also produce the above warning.

Region aggregation and lock declaration (as described above) can be used to suppress these warnings. In general, we can suppress this warning for a field `f` that references objects of class `C` by annotating class `C` with `@ThreadSafe`. This unchecked annotation declares that instances of the class/interface are meant to be “thread safe.” Exactly what is meant by this is presently unspecified; the annotation is intended to encompass non-lock-based protections schemes such as single-threadedness, immutability, and thread confinement. No analysis is performed on classes declared to be thread safe to verify that the implementation is, in fact, thread safe. (For immutable objects you can alternatively use `@Immutable` instead of, or with, `@ThreadSafe`.)

As an example, let us consider the simple rational numbers class below:

```

public class Rational {
    private final int numerator;
    private final int denominator;

    public Rational(int n, int d) {
        numerator = n;
        denominator = d;
    }

    public int getNumerator() { return n; }
    public int getDenominator() { return d; }
}

```

Suppose we have a client class that has a lock-protected reference to a `Rational` object:

```

@RegionLock("Lock is this protects Instance")
public class C {
    ...
    private Rational r;

    public synchronized doStuff() {
        int n = r.getNumerator();
        ...
    }
}

```

The tool is going to generate a warning attached to call `r.getNumerator()` that “Receiver `r` may be a shared unprotected object.” That is, we may be doing something unsafe: accessing the state of the rational object in multiple threads without protection. But in this case, we know that `Rational` objects are immutable, and therefore thread safe, so we can suppress this warning by annotating the `Rational` class:

```

@ThreadSafe
@Immutable
public class Rational {
    ...
}

```

A.1.6 Regions and locks

The lock annotation `@RegionLock` actually associates a lock with a region. A region is a named, hierarchical abstraction of state. All fields are regions, and thus a region is a named, extensible set of fields. Using annotations, the programmer can declare new abstract regions as members of a class, and then associate different locks with different regions. The state of an object may thus be partitioned into multiple abstract regions, each protected by a different lock, enabling concurrent access to different segments of the object’s state.

Declaring new regions

New regions are declared by annotating a class with

```
@Region("visibility Region extends Parent")
```

which declares a new region *Region* which is a subregion of *Parent*. A region may be declared to be **static**, and may have any of the standard Java visibility modifiers (or none of them). A **static** region must extend another **static** region; non-**static** regions may extend **static** regions. If no parent region is specified, **Instance** is used for non-**static** regions; the root region—the **static** region **All**—is used for **static** regions. Java only allows a single annotation of each type to be used at time, so to declare more than one region on a class, the **@Regions** annotation is used:

```
@Regions({
    // region with default visibility, extends Instance
    @Region("region1"),
    // static private region, extends All
    @Region("private static region2"),
    // public instance region
    @Region("public region3 extends region2")
})
```

Unless otherwise specified, a **static** field is in the region **All**, and a non-**static** field is in the region **Instance**. To place a field in a user-declared region, the **@InRegion** annotation is used:

```
@Region("protected NewRegion")
public class C {
    @InRegion("NewRegion")
    private int f;
    ...
}
```

As with abstract regions, a **static** field may only be placed into a **static** region. Non-**static** fields may be placed into either **static** or non-**static** regions.

Associating locks with regions

The general form of the **@RegionLock** annotation is

```
@RegionLock("LockName is Lock protects Region")
```

where *LockName* is a programmer-declared name for the lock, *Lock* is a reference to the lock object, and *Region* is the name of a region. More specifically, *Lock* must be one of the following:

- **this**
- A field declared in the class being annotated or an ancestor of the class being annotated that is visible within the class, for example, a protected field from an ancestor.
- **class**
- A field of an outer class.

When *Lock* is **this**, the object itself is acquired to protect the state. When *Lock* is a field, the field must refer to an object. The field must be **final**: otherwise the lock object referenced by the field could change. The field may be **static** or instance. When *Lock* is **class**, the unique **Class** object referenced by the static pseudo-field **class** must be acquired. (This is the object that is locked by **static synchronized** methods.)

If a class is declared inside class **Outer**, and wants to declare that the field **f** of the **Outer** instance that is the container for the inner class's instance protects a region of the inner class, then the lock reference is given by **Outer.this.f**. Although the tool supports declaring locks that are the fields of outer classes, it is not presently possible to verify their correct use. This is because of deficiencies in both our internal representation and with Java syntax. Given a variable **v** that refers to an instance of a non-**static** inner class, there is no syntactic expression that evaluates to the "outer" object of that instance, that is, the object referenced by **o** in the expression **o.new Inner()**. We allow the declaration of such locks, even though they cannot be checked by the tool, because we have encountered them in real-world code and it is helpful to be able to document the design intent.

A region may only be associated with a lock declared in a class **C** if the region does not contain any fields from superclasses of **C**. This is trivially true if the region is declared in **C**. The region may have abstract sub-regions, but they also cannot contain any fields. Specifically, the region (and any of its subregions) cannot contain any fields, when considered from the point of view of the class in which the lock declaration appears. (Indeed, otherwise you could never associate a lock with a region.) This restriction exists to prevent unsoundness. Suppose class **C** declares a region **R** and populates it with field **f**. Suppose class **D** extends **C**, adds field **g** to region **R**, and also associates **R** with a lock. The problem is, in contexts where a **D** object is viewed as a **C**, such as when a **D** object is passed a method with a **C** parameter, analysis cannot enforce **D**'s locking policy. Thus, fields in **R** that should be protected might not be, and the analysis would not complain.

We allow the protection of an empty region to be delayed because no state can be accessed through that region by "unprotected" superclasses. Any actual code that accesses state in that region must access the region through a subclass that does know about the protection, and therefore analysis can enforce the protection.

A **static** region must be protected by a **static** field or by **class**.

A region may be associated with a lock only if none of its ancestors are associated with a lock. This prevents a region from being protected by multiple locks.

A.1.7 Intrinsic or **java.util.concurrent** locks

The tool understands the semantics of both intrinsic Java locks acquired using the **synchronized** block and **java.util.concurrent** locks, including **ReadWriteLock**. The tool determines which semantics to enforce based on the object referenced by the *Lock* portion of each **@RegionLock** annotation. There are three cases:

1. The class of the referenced lock descends from **java.util.concurrent.locks.Lock**.
2. The class of the referenced lock descends from **java.util.concurrent.locks.ReadWriteLock**.
3. The class of the referenced lock is any other class.

If the class of the lock is not `Lock` or `ReadWriteLock`, then the tool considers the lock to be an intrinsic lock. It considers the lock to be acquired when it is referenced in a `synchronized` block, and to be released at the end of the block.

If the class of the lock object descends from `Lock`, then the tool understands that the lock is acquired by calling one of `lock`, `lockInterruptibly`, or `tryLock`, and released by calling `unlock`. It checks that each lock acquisition must have a subsequent lock release, and that each lock release has a prior lock acquisition. The tool does not specifically enforce the pattern

```
myLock.lock();
try {
    // do stuff
} finally {
    myLock.unlock();
}
```

but, in practice, this pattern must be used to satisfy the analysis due to the bidirectional matching of lock acquisitions and releases.

If the class of the lock object descends from `ReadWriteLock` then the tool understands that writes to the protected region require the write lock to be held, and that reads from the protected region require either the read or the write lock to be held. The write lock is retrieved using the method `writeLock` and the read lock is retrieved using the method `readLock`. For example, the tool finds that the following class is consistent with its annotations:

```
@RegionLock("L is lock protects f")
public class C {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private int f;

    public void set(final int v) {
        lock.writeLock().lock();
        try {
            f = v;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int get() {
        lock.readLock().lock();
        try {
            return f;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

The tool recognizes a common shorthand used with `ReadWriteLocks`: the caching of the individual read and write locks in additional `final` fields of the class that declares the `ReadWriteLock`. For example

```

@RegionLock("L is lock protects f")
public class CC {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock read = lock.readLock();
    private final Lock write = lock.writeLock();
    private int f;

    public void set(final int v) {
        write.lock();
        try {
            f = v;
        } finally {
            write.unlock();
        }
    }

    public int get() {
        read.lock();
        try {
            return f;
        } finally {
            read.unlock();
        }
    }
}

```

This works with both `static` and instance locks. If the original lock field is `static`, the cache fields must also be `static`.

A.1.8 Declaring multiple locks

Multiple locks are declared as members of a class my using the `@RegionLocks` annotation:

```

@Regions({
    @Region("public Location"),
    @Region("public Appearance")
})
@RegionLocks({
    @RegionLock("LocationLock is this protects Location"),
    @RegionLock("AppearanceLock is appLock protects Appearance")
})
public class Sprite {
    protected static Object appLock = new Object();
    ...
}

```

A.1.9 Returning locks

A method may be declared to return a particular lock using the `@ReturnsLock` annotation. This allows an implementation to provide access to a lock object without revealing how that lock is “implemented.” That is, the identity of the field that refers to the lock is kept hidden by the implementation, although the lock object is made accessible to clients. The `@ReturnsLock` annotation is fully checked: it is checked that the method actually returns the

object representing the lock it says it returns. The following code fragment shows an example use of the `@ReturnsLock` annotation.

```
@Region("protected DataRegion")
@RegionLock("DataLock is lock protects DataRegion")
public class C {
    // NOTE: field is private
    private final Object lock = new Object();

    @ReturnsLock("DataLock")
    protected Object getDataLock() {
        return lock;
    }
    ...
    public void doSomething() {
        synchronized (getDataLock()) {
            // Access DataRegion
        }
    }
}
```

An example of this sort of thing in production code is the method `getTreeLock()` in the library class `java.awt.Container`.

A.1.10 Policy locks

Sometimes there is no obvious state to associate with a lock. That is, a lock is being used to enforce a higher-level invariant that requires a section of code to execute atomically with respect to some other section of code. We call locks used for such a purpose *policy locks*. They can be declared using the class annotation

```
@PolicyLock("LockName is Lock")
```

This annotation is similar to the `@RegionLock` annotation except that it does not associate the lock with any particular region of state. The tool does not provide any verification about the uses of policy locks. The annotation is primarily used to document the intent behind the lock, and to suppress tool warnings about a particular lock object being an unknown lock.

One common use for policy locks is to enforce an “initialize once” invariant. Consider this example from `java.util.logging.Logger`:

```
@PolicyLock("InitLock is class")
public class Logger {
    ...
    public static synchronized Logger getLogger(String name) {
        LogManager manager = LogManager.getLogManager();
        Logger result = manager.getLogger(name);
        if (result == null) {
            result = new Logger(name, null);
            manager.addLogger(result);
            result = manager.getLogger(name);
        }
        return result;
    }
}
```

```
}  
}
```

In this example the `static` method does not change any state directly; it is difficult so say what state the lock `Logger.class` is protecting. This is because the lock is ensuring that the method `getLogger` executes atomically with respect to itself. If two threads were allowed to simultaneously execute the method it would be possible to create two new `Logger` objects with the same name, but only one of them would be registered in the global log registry. This would cause problems later on during the use of the loggers. The synchronization ensures that only one `Logger` object is ever created for any given name.

A.2 Method effects

Regions provide an abstract way to name the state of an object. The effects of a method—the state read and written during the execution of that method—may be expressed in terms of regions. Effects are useful when determining whether code can be reordered by a refactoring, and are also necessary to support the analyses that verify the `@Unique` and `@RegionLock` annotations.

Declaring effects

The effects of a constructor or method are declared using the `@RegionEffects` annotation:

```
@RegionEffects("reads readTarget1, ... ; writes writeTarget1, ... ")
```

Specifically, the annotation contains reads and writes clauses that each have a list of one or more targets. The reads clause describes the state that may be read by the method/constructor; the writes clause describes the state that may be read or written by the method/constructor. Because writing includes reading, there is no need to list a target in the reads clause if its state is already described in the writes clause.

Both the reads and the writes clauses are optional: to indicate that there are no effects use `@RegionEffects("none")`. An unannotated method is assumed to have the annotation `@RegionEffects("writes All")` which declares that the method could read from or write to anything in the heap. A target is an extrinsic syntactic mechanism to name references to regions, and can be one of

- **Region** or **this:Region**. **Region** is a region of the class containing the method. The method affects the named region of the receiver object.
- **param:Region**. **param** is a parameter of the method that references an object. **Region** is a region of the class of **param**'s type. The method affects the named region of the object referenced by **param** at the start of the method's execution.
- **pkg.C.this:Region**. **pkg.C** is an “outer class” of the class that contains the annotated method. That is, the method being annotated is in class **D**, and **D** is an inner class of **C**. **Region** is a region of class **pkg.C**. The method affects the named region of the given outer class receiver.
- **any(pkg.C):Region**. **pkg.C** is a class name and **Region** is a region of **pkg.C**. This target indicates that the method affects the given region of any object of class **pkg.C**.
- **pkg.C:Region**. **Region** is a **static** region of class **pkg.C**. The method affects the given **static** region.

The analysis checks that the actual effects of the method implementation are no greater than its declared effects. There are several fine points to this:

- Uses of **final** fields produce no effects.
- Effects on local variables are not visible outside of a method/constructor.

- Effects on objects created within a method are not visible outside of a method.
- Constructors do not have to report effects on the Instance region of the newly constructed object.
- Region aggregation (described below) is taken into account.

Here is a simple “variable” class with effects annotations:

```
@Region("public Value")
public class Var {
    @InRegion("Value")
    private int value;

    @RegionEffects("none")
    public Var(int v) {
        value = v;
    }

    @RegionEffects("reads Value")
    public int getValue() {
        return value;
    }

    @RegionEffects("writes Value")
    public void setValue(int v) {
        value = v;
    }
}
```

A.2.1 Effects and constructors

A constructor that accesses state protected by a locking model may also be verified by checking whether the declared write effects of the constructor are included in the effect `@RegionEffect("writes Instance")`. The constructor must also be annotated with a declaration that it does not start any threads: `@Starts("nothing")` (see Section A.4 for further information about this annotation). Such a constructor cannot pass a reference to the new object to a preexisting thread because it does not write to any objects that existed prior to the invocation of the constructor. It can write a reference to the new object to other objects created during execution of the constructor, but because it cannot start any threads, such a reference cannot be read by another thread.

Here is an example, which, was used to specify and verify much of Doug Lea’s `util.concurrent` library [73] (for which `@Unique("return")` or `@Borrowed("this")` could not be used because of aliasing of `this` into a private field):

```
@Region("public Variable")
@RegionLock("VarLock is lock_ protects Variable")
public class SynchronizedVariable implements Executor {
    protected final Object lock_;

    @RegionEffects("writes nothing")
    @Starts("nothing")
    public SynchronizedVariable() { lock_ = this; }
```

```
    ...  
}  
  
public class SynchronizedLong extends SynchronizedVariable  
    implements Comparable, Cloneable {  
    @InRegion("Variable")  
    protected long value_  
  
    @RegionEffects("writes nothing")  
    @Starts("nothing")  
    public SynchronizedLong(long initialValue) {  
        super();  
        value_ = initialValue;  
    }  
    ...  
}
```

A.3 Unshared fields

Any reference-typed field, not just arrays, can be declared to be `@Unique`. Furthermore, state aggregation allows any region of the uniquely referenced object to be aggregated into a region of the referring object (subject to certain well-formedness rules that ensure the region hierarchy is preserved).

A.3.1 Borrowed references

When an object is passed as a parameter to a method, an alias to that object is created. Thus, if strictly enforced, a unique field can never be passed as a parameter to a method, even as the receiver! But if a method is known to not create any additional aliases to the object, then a unique field may safely be passed as a parameter because it is guaranteed that the method will restore the uniqueness of the field. However, the method is not allowed to directly or indirectly make use of the unique field used as the parameter because the field is not unique within the dynamic scope of the method [21]. A parameter (including the receiver) is declared to be borrowed by annotating the parameter as `@Borrowed`. To declare that the receiver is borrowed, we annotate the method with `@Borrowed("this")`. To declare that a constructor does not alias the object under construction, we annotation it with either `@Unique("return")` or `@Borrowed("this")` (which are defined to be equivalent for constructors).

Consider method `C.copyInternalArray()`:

```
public class C {
    private Object[] myArray;
    ...
    @Borrowed("this")
    public void copyInternalArray(@Borrowed Object[] array) {
        for (int i = 0; i < array.length; i++) {
            array[i] = this.myArray[i];
        }
    }
}
```

Because the method declares that its receiver is borrowed, it may be invoked on `C` objects referenced through `@Unique` fields. It may also be passed references to arrays referenced by `@Unique` fields. Here it is easy to see that no aliases to `this` or to `array` are created, but, in general, this is a property that is easily violated, and a separate set of analyses from those used to verify locking are used to check that `@Unique` fields and `@Borrowed` variables are used correctly.

A.3.2 Supporting borrowed with method effects

As explained above, when a method is passed, the value of a unique field as the actual to a borrowed parameter, the method is not allowed to access the unique field. Analysis looks to the effects of the methods to determine if the method could possibly read the forbidden field. Thus, if a method has borrowed parameters, it is usually necessary to declare the methods effects as well.

Consider the class `Var`:

```
class Var {
    private int value = 0;

    @Borrowed("this")
    public void set(int v) { value = v; }

    @Borrowed("this")
    public int get() { return v; }
}
```

It's obvious that we can declare the receiver to be borrowed for the two methods. If we never actually use a unique field as the receiver, then we do not need to declare the effects of the methods:

```
class VarClient {
    private Var v1 = new Var();
    private Var v2 = new Var();

    public void doStuff() {
        v1.set(1);
        v2.set(2);
        ...
        v1.set(v2.get()+3);
    }
}
```

If instead field `v1` were annotated with `@Unique`, then analysis would need to know that it is not possible for `set()` to read the field `v1` when `v1` is used as the receiver. Here it is obvious that it cannot, but in cases where the invoked method retrieves objects from collections or other global object pools, it is not so clear. We must explicitly declare the effects of `set()` to allow the uniqueness analysis to verify the call (we also declare the effects of `get()` for completeness):

```
class Var {
    private int value = 0;

    @Borrowed("this")
    @RegionEffects("writes this:Instance")
    public void set(int v) { value = v; }

    @Borrowed("this")
    @RegionEffects("reads this:Instance")
    public int get() { return v; }
}
```

A.4 Thread effects

Thread effects can be used to verify that method or constructor does not start any threads. The `@Starts("nothing")` promise states that the method never causes a new thread of execution to start. Consider the example:

```
class C {
    @Starts("nothing")
    public C() {
        m();
        ...
    }

    @Starts("nothing")
    private void m() { ... }
}
```

The code above illustrates the modular nature of the `@Starts("nothing")` promise. Because the constructor promises `@Starts("nothing")` so must the `m` method and the constructor for `Object` (the superclass of `C`). We assume that `java.lang.Object` has been annotated (via XML) so that class `C` will verify.

A.5 Scoped promises

Scoped promises are promises that act on other promises within a static scope of code. Two types of scoped promises are supported by the tool: `@Promise` to avoid tedious user annotation and `@Assume` to support team modeling. Note that scoped promises are still experimental in JSure, however, it is possible to use `@Promise` in models. We will not describe `@Assume` any further.

The simplest example of the use of `@Promise` is to change the default for a class. So instead of writing

```
class Example {
    private int value = 0;

    @Borrowed("this")
    public void set(int v) { value = v; }

    @Borrowed("this")
    public int get() { return v; }
}
```

you would write

```
@Promise("@Borrowed(this)")
class Example {
    private int value = 0;

    public void set(int v) { value = v; }

    public int get() { return v; }
}
```

Notice that the payload promise within the scoped promise has the same syntax as the original annotation except that the quotation marks are removed. So `@Borrowed("this")` is changed to `@Borrowed(this)` when used as the payload for a scoped promise.

In the form shown in the example above `@Promise` places the payload promise on every declaration in the class where the payload promise makes sense. In the example above, this is all methods and constructors within the `Example` class. It is possible to be more specific by providing an explicit target.

```
@Promises({
    @Promise("@Starts(nothing) for new(**)"),
    @Promise("@Unique(return) for new(**)"),
    @Promise("@Borrowed(this) for get*(**)"),
})
class Example2 {
    ...
}
```

In the example above a `@Starts("nothing")` and `@Unique("return")` annotation are placed on all the constructors in the `Example2` class using the explicit target `new(**)`. A

`@Borrowed("this")` annotation is placed on all methods within the `Example2` class that have names that start with “`get`” (e.g., `getValue()`) and have any number of parameters of any type.

A.6 Programmer vouches

It is possible for a programmer to vouch for any inconsistent analysis result within a scope of code. This is done with the `@Vouch` annotation. The scope of this annotation matches the scope of the declaration where the annotation appears. This means that any inconsistent result within that scope will be changed to a consistent result. Its use is for documentation and to quiet overly conservative analysis results.

Use of the `@Vouch` annotation is trusted, *i.e.*, it is not verified by analysis. The annotation requires a brief programmer explanation for the vouch being made.

In the example code below an `init` method is used to set state, perhaps due to an API restriction about using constructors, and then `CentralControl` instances are safely published. An `@Vouch` annotation is used to explain that the `init` method needs to be an exception to the lock policy.

```
@Region("private ControlRegion")
@RegionLock("ControlLock is lock protects ControlRegion")
public class CentralControl {

    private final Object lock = new Object();

    @InRegion("ControlRegion")
    private String f_id;

    @Vouch("Instances are thread confined until after init(String) is called.")
    public void init(String id) {
        f_id = id;
    }

    public String getId() {
        synchronized (lock) {
            return f_id;
        }
    }

    public void setId(String value) {
        synchronized (lock) {
            f_id = value;
        }
    }
}
```

In the example code below a `@Vouch` annotation is used to explain that the `SmokeTest` class is test code that is intended to have inconsistent verification results.

```
@Vouch("Test code that is intentionally inconsistent with models")
public class SmokeTest extends ... {
    ...
}
```


Extended Backus-Naur form

This appendix presents a concise overview of the extended Backus-Naur Form (XBNF) syntax notation presented by Muchnick in [83]. In XBNF terminals are written in **typewriter** font and nonterminals are written in *italic* font. Nonterminals are written using the same case conventions as Java classes (i.e., *MethodDecPat*). A production consists of a nonterminal followed by “ \rightarrow ”. The symbol “ ϵ ” represents the empty string of characters. The XBNF operators are listed in the table below.

Symbol	Meaning
	Separates alternatives
{ }	Grouping
[]	Optional
*	Zero or more repetitions
+	One or more repetitions
\bowtie	One or more repetitions of the left operand separated by occurrences of the right operand

The operators “*”, “+” and “ \bowtie ” all have higher precedence than concatenation which has higher precedence than “[”. The curly braces and square brackets act as grouping operators. All operators are written in the ordinary text font. If the same symbols appear in **typewriter** font then they are terminal symbols.

As an example consider a simplified Java method declaration (no modifiers, throws clause, or body) assuming *Identifier*, *TypeName*, and *MethodName* are properly defined for Java.

$$\begin{aligned} \textit{FormalParams} &\rightarrow \{ \textit{TypeName Identifier} \} \bowtie , \\ \textit{MethodDec} &\rightarrow \{ \textbf{void} \mid \textit{TypeName} \} \textit{MethodName} \\ &\quad ([\textit{FormalParams}]) ; \end{aligned}$$

Glossary

This appendix consolidates and defines noteworthy terms and acronyms used within this dissertation. Definitions are followed by Internet links to further information.

abductive reasoning Abduction is a method of logical inference introduced by Charles Sanders Peirce which comes prior to induction and deduction for which the colloquial name is to have a *hunch*. Abductive reasoning starts when an inquirer considers of a set of seemingly unrelated facts, armed with an intuition that they are somehow connected. The term abduction is commonly presumed to mean the same thing as hypothesis; however, an abduction is actually the process of inference that produces a hypothesis as its end result. http://en.wikipedia.org/wiki/Abductive_reasoning

AST Abstract Syntax Tree is a tree representation of the abstract, *i.e.*, simplified, syntax of a program used for semantic program analysis. http://en.wikipedia.org/wiki/Abstract_syntax_tree

analysis-based verification An automatic property-oriented proof-based program verification approach. Analysis-based verification enables assurance of useful mechanical properties about programs with respect to programmer provided models of design intent, *i.e.*, specifications, expressed by promises about the program. The dominant design consideration for any analysis-based verification system is adaptability by practicing programmers. JSure is prototype tool implemented within the Eclipse Java IDE that performs analysis-based verification.

dependent Dependent drops in drop-sea depend upon evidence, for their truth, of deponent drops allowing drop-sea to serve a truth maintenance role.

deponent A deponent is defined to be one who gives evidence. Deponent drops in drop-sea provide evidence for the truth of dependent drops allowing drop-sea to serve a truth maintenance role.

disjunctive normal form In boolean logic, a disjunctive normal form (DNF) is a standardization (or normalization) of a logical formula which is a disjunction of conjunctive clauses. As a normal form, it is useful in automated theorem proving. A logical formula is considered to be in DNF if and only if it is a disjunction of one or more conjunctions of one or more literals. http://en.wikipedia.org/wiki/Disjunctive_normal_form

drop-sea The proof management system use by our JSure prototype tool. Drop-sea manages the results reported by “plug-in” program analyses and automates the proof calculus developed in Chapter 2 to produce verification results based upon these findings.

Eclipse Java IDE The Eclipse Java Integrated Development Environment is an open source Java software development environment. JSure is a “plug-in” to this tool. Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. <http://www.eclipse.org/>

EJB Enterprise JavaBeans is the server-side component architecture for Java Platform, Enterprise Edition (Java EE). EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology. <http://java.sun.com/products/ejb/>

eAST Eclipse-based Java Abstract Syntax Tree. The AST produced by the Eclipse Java IDE for program analysis and refactoring. This tree “leans” toward a concrete syntax and can parse many illegal Java compilation units (*e.g.*, programs that will not type check).

fAST Fluid IR-based Java Abstract Syntax Tree. A Java AST represented in the Fluid IR designed for program analysis. This tree is very abstract to simplify analysis and will not accept illegal Java compilation units.

Fluid project Fluid is a research project led by Dr. William L. Scherlis at Carnegie Mellon University (with collaboration from a team led by Dr. John Boyland at the University of Wisconsin–Milwaukee). The project is focused on creating practicable tools for programmers to assure and evolve real programs. The work focuses on “mechanical” program properties that tend to defy traditional testing and inspection regimes. <http://www.fluid.cs.cmu.edu/>

Fluid IR The Fluid Internal Representation models general purpose data using an enhanced version of the standard ternary representation: unique identifiers, attributes, and values. The enhancements made to the standard ternary representation include (1) ultra-fine-grained tree-structured versioning, (2) abstraction to structured entities such as trees and directed graphs, and (3) persistence. The Fluid IR is used within the JSure tool to represent programs as a “forest” of fASTs, it is also used to model flow graphs of program control flow, bindings from a use to a definition/declaration, annotations of programmer design intent, analysis results, *etc.*.

Hasse diagram A Hasse diagram is a simple picture of a finite partially ordered set, forming a drawing of the transitive reduction of the partial order. A point is drawn for each element of the poset, and line segments are drawn between these points with an implied upward orientation. http://en.wikipedia.org/wiki/Hasse_diagram <http://mathworld.wolfram.com/HasseDiagram.html>

IDE An Integrated Development Environment is a software program that provides comprehensive facilities to programmers for software development tasks. An IDE normally consists of: a source code editor, a compiler and/or and interpreter, build automation tools, a debugger, and an integrated version control system. http://en.wikipedia.org/wiki/Integrated_development_environment

-
- JAR** a Java ARchive file aggregates many files into one. Software developers generally use `.jar` files to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.) JAR files build on the ZIP file format. Computer users can create or extract JAR files using the `jar` command that comes with a JDK. They can also use zip tools to do so. [http://en.wikipedia.org/wiki/JAR_\(file_format\)](http://en.wikipedia.org/wiki/JAR_(file_format))
- JDK** The Java Development Kit is a Sun Microsystems product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java SDK. On 17 November 2006, Sun announced that it would be released under the GNU General Public License (GPL), thus making it free software. This happened in large part on 8 May 2007 and the source code was contributed to the OpenJDK. <http://openjdk.java.net/>
- Java EE** Java Platform, Enterprise Edition is an industry standard (with multiple commercial and open source implementations) for enterprise Java computing. <http://java.sun.com/javaee/>
- JSure** A prototype tool implemented within the Eclipse Java IDE that performs analysis-based verification.
- KSLOC** 1,000 source lines of code, or kilo-source lines of code. KSLOC is a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code. http://en.wikipedia.org/wiki/Source_lines_of_code
- proof management** The manipulation of formal proofs and proof fragments, *i.e.*, lemmas, as data structures.
- proposed promise** A specification, including an assertion and code location, for a promise. These promises typically are used to express preconditions for analysis results. A proposed promise may or may not exist as a real promise.
- promise** A program annotation mechanism to express extra-linguistic design intent within code for use by automatic analysis tools introduced by Chan, Boyland, and Scherlis in [26]. Each promise within a program expresses an assertion about that program's behavior. Promises provide a concrete specification language for analysis-based verification. Promises have precise meaning and are modular in the sense that the implementation constraint on a program by any promise is limited in scope—avoiding a complete program (*i.e.*, closed-world) assumption which would limit the utility of analysis-based verification.
- promise logic** An intuitionistic propositional logic where a proposition represents the consistency of a promise about a program with respect to that program's implementation. Promise logic allows us to symbolically reason about the consistency of promises.
- real promise** A promise explicitly expressed by the programmer. A real promise exists as an annotation to code or as an XML specification of an annotation to code.
- red dot** A visual metaphor used in the JSure user interface to indicate a contingency in the verification of a promise. The red dot is a metaphor that the programmer has signed in blood that the contingency is met with respect to the program being analyzed.

- sound combined analyses** An approach to analysis-based verification through which results of diverse low-level program analyses can be combined in a sound way to yield results of interest to software developers.
- top** (\top) A formula in promise logic that stands for tautology. In logic, a tautology is a formula which is true in every possible interpretation. [http://en.wikipedia.org/wiki/Tautology_\(logic\)](http://en.wikipedia.org/wiki/Tautology_(logic))
- truth maintenance** Maintaining traceability from antecedents to consequents. When an antecedent is removed, the affected consequents can be identified without the necessity to regenerate the proofs.
- util.concurrent** Refers to the `java.util.concurrent` package added via JSR-166 to Java 5. The package contains a library of useful, well-tested, concurrency components that are useful when building concurrent applications in Java. The library was originally released by Doug Lea as `EDU.oswego.cs.dl.util.concurrent` and was described in [73]. While JSR166 has completed and is a now final approved JCP spec, the expert group remains involved in incremental improvements and changes to the `java.util.concurrent` package and related classes and packages. <http://gee.cs.oswego.edu/dl/concurrency-interest/index.html>
- XBNF** The Extended Backus-Naur Form syntax notation presented by Muchnick in [83]. A concise overview of the XBNF syntax notation is presented in Appendix B.
- XML** The Extensible Markup Language is a set of rules for encoding documents electronically. XML's design goals emphasize simplicity, generality, and usability over the Internet. It is a textual data format, with strong support via Unicode for languages of the world. <http://en.wikipedia.org/wiki/XML>

Bibliography

- [1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 682–696, London, UK, 1997. Springer-Verlag.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- [3] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM Press, May 2002.
- [4] Eric Allen. *Bug Patterns in Java*. APress, 2002.
- [5] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37, New York, NY, USA, 2005. ACM.
- [6] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy*, pages 131–147. IEEE Computer Society Press, 2002.
- [7] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA '00*, pages 382–400, 2000.
- [8] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, pages 1–3, New York, January 2002. ACM Press.
- [9] John Barnes. *High Integrity Software: The SPARC Approach to Safety and Security*. Addison-Wesley, 2003.
- [10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 CASSIS (Construction and Analysis of Safe, Secure, and Interoperable Smart devices) International Workshop*, pages 49–69. Springer-Verlag, March 2004.
- [11] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*,

- volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, 2006.
- [12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.
- [13] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.
- [14] Kent Beck and Erich Gamma. JUnit test infected: Programmers love writing tests. <http://www.junit.org>, 2006. Current January 2006.
- [15] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [16] Juan C. Bicarregui. *Proof in VDM: A Practitioner’s Guide*. Springer-Verlag, 1994.
- [17] Juan C. Bicarregui and Sten Agerholm. *Proof in VDM: Case Studies*. Springer-Verlag, 1998.
- [18] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.
- [19] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.
- [20] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.
- [21] John Boyland. The interdependence of effects and uniqueness. In *Workshop on Formal Techniques for Java Programs at ECOOP’01*, June 2001.
- [22] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer-Verlag.
- [23] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using permissions. In *ECOOP 2009 Workshop on Aliasing, Confinement and Ownership*, July 2009.
- [24] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O’Reilly, 2006.
- [25] Richard W. Buskens. Tracking down the most elusive bugs. *Lockheed Martin Advanced Technology Laboratories TechBriefs*, 2(7), December 2007.
- [26] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *18th International Conference on Software Engineering (ICSE’98)*, pages 167–176. IEEECS, 1998.
- [27] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.
- [28] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check FLASH protocol code. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’00)*, pages 59–70. ACM Press, 2000.

- [29] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.
- [30] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22th International Conference on Software Engineering (ICSE'00)*, pages 762–765, 2000.
- [31] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A rational module system for Java — and its applications. In *Proceedings of the 18th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003)*, pages 241–254. ACM Press, 2003.
- [32] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.
- [33] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. Simon & Schuster, 1998.
- [34] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.
- [35] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, May 2005.
- [36] Linda DeMichiel and Michael Keith. *JSR 220: Enterprise Java Beans, Version 3.0: EJB Core Contracts and Requirements*. Sun Microsystems, Inc., 2006.
- [37] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.
- [38] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq SRC, December 1998.
- [39] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [40] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
- [41] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [42] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating System Design & Implementation (OSDI'00)*. USENIX, 2000.

- [43] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th Symposium on Operating System Principles (SOSP'01)*, pages 57–72. ACM Press, 2001.
- [44] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–65, 1979.
- [45] Cormac Flanagan and Stephen H. Freund. Type-based race detection for Java. In *2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232. ACM Press, 2000.
- [46] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE'01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM Press, June 2001.
- [47] Cormac Flanagan, K. Rustan M. Lenio, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation*, pages 234–245, New York, June 2002. ACM Press.
- [48] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium in Applied Mathematics*, 19:19–31, 1967.
- [49] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2nd edition, 1999.
- [50] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.
- [51] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- [52] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition, 2005.
- [53] Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2003.
- [54] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer-Verlag, June 1999.
- [55] Aaron Greenhouse, T.J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Science of Computer Programming*, 56(1), 2005.
- [56] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *24th International Conference on Software Engineering (ICSE'02)*, pages 453–463. ACM Press, May 2002.
- [57] John V. Guttag, James J. Horning, S. J. Garland, A. Modet K. D. Jones, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

- [58] Scott C. Hale. Flashlight: A dynamic detector of shared state, race conditions, and locking models in concurrent Java programs. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, 2006.
- [59] Jeffrey S. Hammond. IDE usage trends for application development and program management professionals. Forrester Research, Inc., February 2008.
- [60] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
- [61] C.A.R. Hoare. Viewpoint retrospective: An axiomatic basis for computer programming. *Communications of the ACM*, 52(10):30–32, 2009.
- [62] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.
- [63] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE'07: Proceedings of the 2007 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14. ACM, 2007.
- [64] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.
- [65] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
- [66] Report on *Information Technology Research, Innovation, and E-Government* by the Committee on Computing & Communications Research to Enable Better Use of Information Technology in Government, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences, National Research Council. The National Academies Press, 2002.
- [67] Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Nijmegen Institute of Computing and Information Sciences*, pages 134–153. Springer-Verlag, 2003.
- [68] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
- [69] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
- [70] S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, 1950.
- [71] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
- [72] Patrick Lardieri. Developing and maintaining DoD software intensive systems: Sides of the same coin? ICSM'06 Keynote Address, 2006.

- [73] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 2nd edition, 2000.
- [74] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.
- [75] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Department of Computer Science Technical Report 98-06u, Iowa State University, April 2003.
- [76] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: Notations and tools supporting detailed design in Java. Department of Computer Science Technical Report 00-15, Iowa State University, August 2000.
- [77] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, October 2000.
- [78] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49, New York, NY, USA, 2003. ACM Press.
- [79] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. John Wiley & Sons, 2nd edition, 1987.
- [80] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the 16th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2001)*, pages 211–222. ACM Press, 2001.
- [81] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
- [82] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2nd edition, 1997.
- [83] Stephen S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [84] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE - specification and design rationale. Technical Report D-58084 Hagen, Fernuniversität Hagen, 1997.
- [85] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, pages 29–36, 1997.
- [86] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [87] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

- [88] H. Penny Nii. Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, 7(4):82–106, 1986.
- [89] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(3):38–53, 1986.
- [90] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: An experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.
- [91] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992.
- [92] David L. Parnas. Designing software for ease of extension and contraction. In *3rd International Conference on Software Engineering (ICSE'78)*, pages 264–277. IEEE Press, 1978.
- [93] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [94] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International Series in Computer Science. Prentice Hall Europe, 2nd edition, 1996.
- [95] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.
- [96] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. MIT Press, 2001.
- [97] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1st edition, 1990.
- [98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
- [99] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Pearson Education, 1996.
- [100] Charles Simonyi and Martin Heller. The Hungarian revolution. *Byte*, 16(8), August 1991.
- [101] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *MSR'06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136. ACM, 2006.
- [102] Mladen Stanojevic, Sanja Vranes, and Dusan Velasevic. Using truth maintenance systems. *IEEE Expert*, pages 46–56, 1994.

- [103] Dean F. Sutherland. *The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2008.
- [104] Dean F. Sutherland, Aaron Greenhouse, and William L. Scherlis. The code of many colors: Relating threads to code and shared state. In *PASTE'02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 77–83. ACM, 2002.
- [105] Dean F. Sutherland and William L. Scherlis. Composable thread coloring. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 233–244. ACM, 2010.
- [106] David A. Wagner and Daniel Dvorak. Testbed analysis of dependability properties using the Fluid software assurance tool. Final report JPL task # 1166, NASA/JPL, 2008.
- [107] Jeannette M. Wing and Chun Gong. Experience with the Larch prover. In *Conference proceedings on Formal methods in software development*, pages 140–143. ACM Press, 1990.
- [108] Stomping out Java “concurrency cockroaches” with SureLogic’s Flashlight and JSure tools. Yahoo! Developer Network Blog: http://developer.yahoo.net/blogs/hadoop/2010/01/stomping_out_java_concurrency_1.html, January 2010.